University of Alberta

Library Release Form

Name of Author: Peter J. Woytiuk

Title of Thesis: HIME: Hierarchical Interactive Modifiable Environments

Degree: Master of Science

Year this Degree Granted: 2001

I am so smart! I am so smart! S-M-R-T.... I mean S-M-A-R-T.
Homer J. Simpson

University of Alberta

HIME: Hierarchical Interactive Modifiable Environments

by

Peter J. Woytiuk ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **HIME: Hierarchical Interactive Modifiable Environments** submitted by Peter J. Woytiuk in partial fulfillment of the requirements for the degree of **Master of Science**

To My Parents, who've always believed in me.

Abstract

We present a system for interacting in real-time with volume data that enables to user to dynamically alter the information contained in the dataset. HIME is a framework that combines voxel data, marching cubes, and octrees which allows for fast collision detection. The volume data is stored in the octree as a collection of voxels, with the original data stored at the leaf level. Each successive level up the tree contains a downsampled version of the previous level creating a hierarchy of varying-resolution models. The voxel data for each given level is polygonized using a simplified form of the marching cubes algorithm. Modifications to the voxel data occur locally, and use a simple protocol to propagate to other levels in the octree. A system of this type can be used to render large-scale MRI and CT study datasets in real-time, and can be extended to perform isosurface exploration for datasets of this type; numerous other applications are also possible.

Acknowledgements

I would like to thank Dr. Benjamin Watson for his patience, understanding, and most of all his input throughout the course of this project – without him HIME would still be some tired scribblings on a dirty whiteboard. I would like to thank Ehud Sharlin for the countless nights he kept me company in the lab, and for all the advice he provided. I would also like to thank BioWare for not firing me for all the times I came in late and left early, in fact quietly turning a blind eye on these events, on account of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Many of today's graphical applications that deal with 3-D objects provide support for reducing the visual fidelity of the scene in order to improve interactivity. This allows the display of a model that, in its original state, could not have been rendered in real-time. An implementation of this type of system often involves a hierarchical structure, with the various levels in the hierarchy representing the model at different levels of visual accuracy. The problem with using this type of system is that the models throughout the hierarchy are static — if the user wishes to make any modifications to the object the process of dynamically updating all levels of the hierarchy would grind the application to a halt, negating the whole point of using the hierarchy in the first place. Often this means that changes are made to a wireframe model, or some tiny sub-section of the model, and then rendering the entire object to see how it was affected by the changes.

What I present is a framework within which modifications may be carried out on large-scale model, all the while preserving both interactivity and fidelity. This framework relies on spatial subdivision through the use of octrees, and polygonization through the use of a modified marching cubes algorithm. Collision detection for modification is based on the representation of the object as a collection of voxels.

1.2 Previous Work

1.2.1 Octrees

Octrees are a method of spatial subdivision, whereby an area is recursively divided into 8 subvolumes [WG90, Sam90]. The basic implementation has each coordinate axis split in half, with the bisection of the splits creating octants of the original space. They are useful in the creation of hierarchical model representations for use in controlling model fidelity.

1.2.2 Marching Cubes

Marching cubes is an algorithm to create isosurfaces from discrete 3-dimensional datasets, usually produced by medical devices such as an MRI machine or a CT scanner. Isosurfaces are specified by an isovalue, where an isovalue specifies the value across which a surface will be created throughout the entire dataset [LC87].

1.2.3 Level of Detail

Level of detail is an area of research that focuses on decreasing the visual fidelity of objects that have been determined to be of less importance than others [Lue97, Gar99]. This importance is most often defined in terms of the screen area a particular object occupies – the less area covered, the lesser the importance. The methods for decreasing a model's visual fidelity are quite varied, and remain an area of ongoing research. A level of detail system allows the interactive display of a model that, in its original form, would have rendered at non-interactive speeds. With models becoming increasingly complex there is a need for better and better systems that can handle these types of models, while still preserving as much of the original detail of the model as possible.

1.2.4 Modifiable Environments

Much of the work done with modifiable environments has been with sculpting applications where the user carves away an object from a block of wood or a lump of clay [WK95, GH91]. There has been some research done with multi-resolution modeling, where portions of a surface are translated, rotated, or scaled using higher-order surfaces to regenerate the surface [ZSS97, KCVS98]. In this context, fidelity control provides a means for specifying the scale of an edit as well as maintaining interactivity. For a system of this type the surface is merely deformed; this is in contrast to HIME which deals with actual 'material' removal. Several processes are involved in this, from collision detection to surface regeneration. Applications other than sculpting are also possible, including digging simulators, medical simulators and games. I present a framework upon which any one of these applications can be built — the example I chose to develop was a mechanical shovel simulator.

1.3 Objectives

The success of this project was to be measured by several criteria:

- **Large data sets** On current graphics systems polygonalizations of 64^3 datasets can be displayed at rates nearing interactive levels. As such, 256^3 was considered to be the minimum size that was to be displayed by the system, with the target size approaching 512^3.

- **Interactivity** The system needs to provide an interactive response to the user. "Interactive" was considered to be anything acheiving a frame rate of 10 frames per second. This would need to be maintained both while maneuvering the object, and while modifying its surface.
- **Visual Quality** Representation of a high-detail dataset (say 512^3) must be better than simply displaying a 64^3 subsampled version.

1.4 Contributions

This system is designed to work with very large datasets, and provides an accurate depiction at interactive levels. Interactive modification of sets this large has never been acheived, much less at this high a framerate. Furthermore, this contribution provides a foundation for a tool allowing interactive exploration of the many isosurfaces within a 3-D dataset. This tool is somewhat outside the initial project's scope, and as such would require some additional work.

1.5 Test Machines

Throughout this document, examples are used to back up certain design decisions made throughout the course of HIME's implementation. These decisions are made either to improve the speed of rendering or the quality of the model approximation. Testing of HIME's results was done on 2 machines: an Octane with 2 175 MHz R10Ks with 1MB cache, 128 MB RAM, and an SI graphics board; and an Onyx2 with several 195 MHz R10Ks with 4 MB cache, 768 MB RAM, and an InfiniteReality2 graphics board. The Octane is referred to as goodwin, while the Onyx2 is referred to as cadomin. Both machines are multi-processor machines, but only one processor was used. A thorough discussion of HIME's performance is given in chapter 5.

Chapter 2

Previous Work

2.1 Level of Detail

2.1.1 Overview

The use of polygonal models is widespread throughout the graphics community, evidenced by the fact that hardware-accelerated functions for specifying and handling polygons have been available for many years. Polygons are extremely useful, as they serve as a sort of lowest common denominator - any type of model specification method be it CSG, high-order surface, or volumetric dataset can be converted with reasonable accuracy to a collection of polygons.

The usefulness of the polygonal standard has lead to unprecedented ease of generating this type of model, and unfortunately, difficulty in rendering them: the largest models are easy to generate but hard to display. This has long been a problem and there have been numerous papers published on the subject [KBGT97, Lue97, Gar99]. Solutions to this problem are split into 2 categories: dynamic methods and static methods.

Dynamic methods [Hop97, ILG96, LE97, XESV97] create a surface approximation on the fly – depending on the current viewpoint of the user a subset of the model is selected as being pertinent and is rendered at a high resolution. The remainder of the model is rendered at lower resolutions, often depending on some form of fall-off based on the distance to the viewpoint.

Static methods generate a coarse approximation offline and render a single model at a time [BW00, GH91]. In a system with a dynamic viewpoint a hierarchy of models may be generated, each at successively smaller resolutions. During run-time the algorithm employs some form of distance metric to decide which of these models to display.

There are numerous methods for creating the coarse approximations used in these LOD systems. Some of the more common approaches have been:

- Culling hidden triangles. Based on the current viewpoint, triangles that are fully occluded by other triangles belonging to any particular object can be removed, thus

reducing the amount of workload placed on the rendering hardware.

- Using a discrete set of models. A given object will switch to a lower resolution model as the distance between it and the camera increases.

- Hierarchical simplification. An object is composed of a series of nodes that can be collapsed or expanded based on some metric, allowing either unimportant or highly-detailed nodes or model regions to be represented by some smaller subset of polygons. The idea is to create a simplified polygonal mesh that matches as closely as possible the original mesh, while reducing the amount of information needing to be rendered.

Each of these presents their own set of problems. Culling hidden triangles is an excellent start, and is simple enough to implement that hardware vendors have begun offering integrated functionality to do so. The problem comes when a large model is rendered at a great distance from the viewpoint. Hundreds, even thousands of polys run through the graphics pipeline only to show up on-screen as a scant few pixels. The same result could be obtained using a small number of polygons. Discrete model sets suffer from a "popping" effect during the transition from one model to another: the sudden switch can be quite obvious. One solution is to have a large set, but this becomes wasteful of often precious memory resources. Lastly, hierarchical simplification methods can be expensive: the model must be subdivided in some fashion, a method for determining which node should be rendered is required, and the collection of nodes needs to be organized in such a manner that a particular node can be fished out of the tree in very little time.

There are many different methods for creating varying levels of surface representations, and is an area in which research still continues. For surveys of existing and potential research see [KBGT97, Lue97, Gar99].

2.1.2 HDS

The work presented in this document draws much from [LE97]. In that work, Luebke describes a method called ***hierarchical dynamic simplification*** or HDS, a system which dynamically retesselates a given scene as the user's viewpoint shifts throughout the scene's space. HDS works by clustering vertices together into a single large data structure, labelled a vertex tree. It is this tree that is dynamically queried to generate the simplified scene. Each node in the tree contains one or more vertices – HDS operates by collapsing all of the vertices within a particular node to a single representative vertex. A node may also be expanded, during which the representative vertex is replaced by the children of that node.

Nodes to be collapsed or expanded are chosen based on their projected size, a process which operates continuously. As the user traverses through the scene, certain nodes will fall below the given size threshold and will be folded into their parent; any degenerate triangles

created by this are removed from the display list. Other nodes will increase in apparent size and if they increase past the threshold they will be unfolded and replaced by their children, causing new triangles to be inserted into the display list. The size threshold is selectable by the user, and may be changed throughout the course of a viewing session allowing the user to control the degree of simplification.

This size threshold is but one criterion Luebke uses to determine which nodes are folded or unfolded each frame; HDS also incorporates a silhouette test and a triangle budget. The silhouette test is designed to improve the visual accuracy of the object's contour, which is of special perceptual importance. Luebke associates with each node in the vertex tree a cone which bounds all of the triangle normals contained by the subtree of that node. This cone is tested against a viewing cone originating from the viewer's position, and encloses the node's bounding sphere. This test can determine whether the node is completely front-facing, completely back-facing, or potentially on the silhouette. Nodes that may be on the silhouette are then tested against a tighter size threshold than those that are front-facing. Note that backfacing nodes may be removed entirely.

The HDS triangle budget places a limit on the number of triangles that may be rendered in a given frame. The system then unfolds nodes until this maximum has been reached. Nodes are placed in a priority queue, and are sorted from greatest to least in order of screen size. The node with the greatest error is unfolded and its children re-inserted into the priority queue. This process iterates until unfolding the top node would exceed the triangle budget.

According to Luebke, HDS operates with adequate speed on small models, no larger than 20,000 triangles. He also claims that exploiting temporal coherence, using visibility information, and parallelizing the algorithm provide a speed increase of two orders of magnitude.

2.2 Marching Cubes

The marching cubes algorithm was conceived to create polygonal isosurfaces from discrete volume datasets [LC87]. The idea is that the data is arranged in a 3-D grid structure, and the algorithm 'marches' through it incrementally. The first use for MC was for extracting surfaces from medical datasets, such as those generated by MRI or PET machines. These machines capture numerous virtual slices of a body, often several dozen, which can be difficult to visualize in 3 dimensions. Marching cubes iterates across these slices, and creates a triangle topology between each slice pair.

At each iteration, the algorithm creates a cube across 2 slices, taking 4 adjacent values from one slice and 4 from the next slice. Each of these 8 corner values (4 in the 2D case) are called ***voxels***, and the cube that they form is called a ***cell***. A voxel is defined to be a

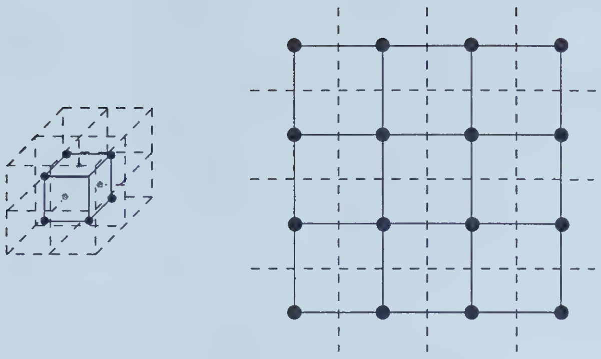


Figure 2.1: Voxels (dashed outline) vs. Cells (solid outline)

sample of the volume in a regularly-spaced 3-D grid; V is used to refer to a single voxel. A cell consists of 8 neighbouring voxels in the voxel grid grouped in a cube configuration; C is used to refer to a single cell. It is these cells that form the basis for the marching cubes algorithm. In figure 2.1 cells are indicated by the solid line and the voxels are indicated by the dashed line - both the 2D and 3D variants are shown. The solid black dots indicate the voxel centres, and it is clear that the cells are formed simply by connecting the dots at rights angles. Note that a given voxel may in turn be shared by up to 8 cells.

It needs to be emphasized that voxels and cells occupy similar amounts of volume, but represent wholly different ideas. A voxel value essentially represents an average for the entire volume it occupies and is placed at the centre of the voxel's space. A cell represents a collection of 8 voxels, defined by connecting the centre points of said 8 voxels, and is used by the marching cubes algorithm to create a polygonal mesh. This differentiation between voxels and cells is paramount to the operation of the HIME algorithm.

Each corner is classified as being "in" or "out" based on the desired value at the surface, called an ***isovalue***. A corner is "in" if its value is greater than or equal to the isovalue, and "out" otherwise. The surface generated using a particular isovalue is called an ***isosurface***, and a set of slices can contain any number of isosurfaces.

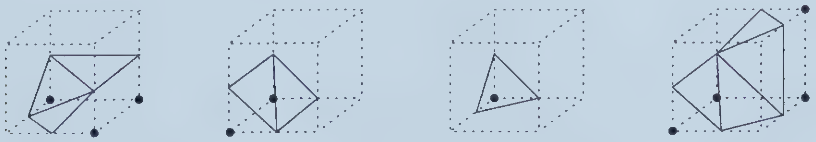


Figure 2.2: Marching cubes cases, with an isovalue of 0.5.

Once all the corners have been classified a bit field is constructed, with one bit for each corner. A given bit is set only if the corresponding corner is labelled as "in". Figure 2.2 shows some of the 14 base MC cases with a surface generated at isovalue 0.5. The black

dots represent voxels labelled as "in", and since the isovalue is 0.5, the vertices are placed at the midpoints of all edges connecting with at most 1 in voxel.

With 8 corners this gives a total number of 256 possible bit fields, which can in turn be used as a key for a table to determine how to polygonize the given cell. Each element in the table lists which edges are intersected by the surface for the current cube configuration. The exact intersection point is determined by interpolating along the intersected edge.

It turns out that these 256 different cases are not unique, in fact the majority of them are simply rotations or reflections of certain 'base' cases. There are 14 base cases in total, and by using symmetry it is trivial to generate the case table.

2.3 Octrees

2.3.1 Overview

Octrees are hierarchical data structures used to represent a subdivision of a given space. How the space is decomposed, and how the final structure is stored, is entirely dependent on the application. A node in the tree will have up to 8 children, with the parent node representing the entirety of it's children. In most octree applications, the tree is used to represent some property of the points within it, with the points being grouped according to this property [Sam90]. As such, there are 2 parts to any octree algorithm:

- The grouping criterion
- The splitting boundary location

The grouping criterion specifies how the elements of the space are to be associated, and ultimately determines if the given octant should be split. If the octant contains disassociated items, it will be further subdivided. In many applications this criterion is the size of the space being subdivided, but in no way is it limited to this. Points can be grouped according to colour, number of neighbours, or some other user-defined value. A common criterion in computer graphics is the number of points within a given space: octants containing any number of points will be subdivided, while those with none at all will not be.

Once an octant has been selected for subdivision, the boundaries for the new octants must be determined before proceeding. The optimal selection process will place the boundaries in such a way that each new octant will contain like objects, or objects that otherwise satisfy the grouping criterion in a similar fashion. Since it is computationally difficult to search each and every octant for 3 splitting planes, some heuristic must be adopted to speed this calculation. There are many different heuristics, again depending on the application, each giving a different octree. Some will create an evenly-balanced octree, useful for maintaining an equal depth across the tree; some will preserve memory, making the tree as compact as

possible; others will create a tree that is ordered in such a way to speed any access to the leaf nodes [Sam90].

The nature of octrees and the fact that they create cells that are most often quadrilateral in shape dovetails nicely with the marching cubes algorithm. One of the biggest problems with MC is that the majority of processing time is spent traversing empty voxels. Much research has been done on combining MC with an octree structure to reduce this time, as well as approximate the given object [GWW86, MSS94, SFYC96]. Other work has been done that combines octrees with the different surface extraction and rendering techniques [Lev90], but employ other methods of rasterization for display.

The actual data stored by the octree is as varied as the algorithms themselves. The octree is used to encode coherence in the volume data, whether that be hierarchical spatial enumeration [Lev90], bounding surfaces [GWW86], or the actual polygonal representation of the surface approximation [MSS94, SFYC96].

2.3.2 BONO: Branch On Need Octrees

The BONO algorithm is a method of creating octrees that are space-efficient while preserving the integrity of the data [WG90]. This allows the tree to be as compact as possible while still allowing for easy access to any of the nodes.

A full octree is one in which each node on each level has exactly 8 children. If we subdivide based on volume, a full octree is only possible if the volume's resolution is the same power of 2 along each axis since ideally each axis would be split in half at each iteration. It is very difficult to impose this type of limit on the resolutions that an algorithm can accept; as a solution, BONO makes the distinction between a ***conceptual region*** and an ***actual region*** in subdividing the world-space. In defining these regions, the ***range*** in each of the x, y, z directions is the difference between the upper and lower limits of the given region, with the ***range vector*** being the triple of x, y, z ranges. A further element to the BONO algorithm is that both the range and range vector are numbered as starting from 0. The conceptual region corresponds to a volume having a range vector made up of the smallest powers of 2 that completely enclose the given volume, regardless of whether the volume extends out that far; the actual region is simply the original volume. It is the conceptual region that is used for subdivision, and as such it is possible to obtain subdivided regions that are completely empty. This results in a few large empty volumes near the top of the tree contained in a few nodes, which results in fewer pointers than having a large number of small empty volumes scattered amongst those nodes near the leaf level.

We use as a running example a $320 * 320 * 40$ volume: the conceptual region for the root would be a $512 * 512 * 512$ volume, while the actual region would be $320 * 320 * 40$. The range vector for the root node of this volume would be:

Axis	Range in Binary	Range in Decimal
x	100111111	319
y	100111111	319
z	000100111	39

Table 2.1: Range vector example for the root node

When dividing a parent node, the directions that branch are those that have a 1 bit in the leftmost position when all the ranges are written with the same number of bits. In the example given in table 2.1 the x and y directions would branch.

If a node is to be split, its children are referred to as "lower" and "upper" along each of the branching axes. Each child has an associated number from 0-7 which describes the particular subregion of its parent that it covers, and serves to identify the given child. This number, when written in binary, uses the zyx convention within BONO: if the z bit is set, the child covers the octant that is "upper" in z; if it is 0 it covers the "lower" region in z. The y and x bits are interpreted similarly. Using this convention, the root node in our example will have the 000, 001, 010, and 011 children filled, with the remaining children being empty.

Once it has been determined that a node needs to be split, range vectors need to be calculated for all of the non-empty children. The actual range for a child in the upper region of a split axis is obtained by simply removing the leading 1 of the original range for that axis. The range of the lower child is a bitstring of 1s, one shorter than the parent's original range. Recall that the root node given in table 2.1 was to be split along the x and y axes; following these rules, the range vector of the 001 child for the root would be:

Axis	Range in Binary	Range in Decimal
x	00111111	63
y	11111111	255
z	00100111	39

Table 2.2: Range vector for the 001 child of the root node

Continuing with this, the resulting conceptual and actual regions for the children along the cardinal axes are listed in table 2.3.

The number of bits in the range vector of the root node, written with the same number of bits, equals the height of the entire tree. For each successive level, its height in the tree is indicated in the same fashion, with the leaves being stored at height 1 and the data at height 0.

A basic implementation of this form of octree is extremely efficient - on goodwin, obtaining a leaf node from an 8-level tree on average occurs in less than 10 μs.

Node	Conceptual Region			Actual Region		
	x	y	z	x	y	z
root	0–511	0–511	0–511	0–319	0–319	0–39
child 000	0–255	0–255	0–255	0–255	0–255	0–39
child 001	256-511	0–255	0–255	256-319	0–255	0–39
child 010	0–255	256-511	0–255	0–255	256-319	0–39
child 100	0–255	0–255	256-511	0–255	0–255	*empty*
...		...			...	

Table 2.3: Conceptual & actual regions for a $320 * 320 * 40$ volume

2.4 Collision

Determining surface interpenetration in real time is a challenging problem, one that continues to be at the forefront of graphics research today [GLM96, CLMP95]. A voxel-based approach lends itself well not only to collision detection, but to surface modification as well [GH91, WK95, Mel98]. A surface can be discretized and stored in a 3-D array as a cloud of interior points. Collision detection is then trivial – the colliding object's volume is discretized and used as a lookup into the surface interior array. If an interior value is found at that element a collision occurred and surface modification may occur.

This type of approach requires a bit more thought when extracting the new surface from the modified surface interior value. Since only a small portion of the surface has been changed, there is no sense running MC across the entire surface. An incremental marching cubes algorithm is much more efficient and is exactly what Galyean & Hughes implemented to speed up their scuplting program. Unfortunately, their volumetric modeller was severely limited, mostly due to their research having been conducted nearly 10 years ago on hardware that would be considered archaic by today's standards. Though research has continued in the area [MPT99] the basic idea remains the same: discretize the colliding objects to aid in locating surface interpenetration. The work done in [MPT99] is not a modeller, it simply detects collisions between a dynamic object and a static scene. The dynamic object is controlled by a 6-DOF haptic device, and the system is able to maintin a 1 KHz refresh rate while calculating a reaction force and torque at each contact point. These calculations and refresh rate are required by the haptic device in order to provide accurate feedback. This demonstrates that while being expensive, collision detection algorithms can still be made to operate in extremely interactive applications and give reasonable results.

Chapter 3

HIME

3.1 Basic Algorithm

The idea behind HIME is to use octrees for area subdivision & representation, a variation of marching cubes to construct isosurface geometry, with voxel-based collision detection to interactively modify the surface. Integrating octrees into HIME addresses the biggest problem with classic marching cubes: the majority of the processing time is spent traversing cells that contain similar information or no information at all, for example volumes containing planar surfaces. Because the "marching steps", or cells, tend to be quite small relative to the volume, processing large datasets can be quite demanding and certainly non-interactive. If there is little information to process in this space then the majority of time is wasted. If the space was subdivided based on local content, as with an octree, by querying this tree the marching cubes algorithm can determine whether or not to approximate a certain space, or even to skip past it if the sub-area is empty.

An octree also allows spatial information to be stored at each level of the tree, instead of just at the leaf level. This enables the storage of varying levels of surface approximation which can be used to speed rendering. The system requires 2 elements to take advantage of this functionality: a method to create each successive surface approximation, and a form of detail control to decide which nodes of the tree get rendered.

3.1.1 Surface Approximation

In order to populate each level in the octree, some form of surface approximation needs to be defined. The base level in the tree is formed from the initial voxel data, with each successive node up the tree being formed by creating a single large voxel from 8 smaller voxels; we currently approximate with simple averaging. This continues all the way up to the root of the tree, which is represented by a single voxel that encompasses the entire volume. The result is a tree of successively larger, or coarser, voxels. The root of the octree is considered to be level 0 with the leaves being at level L. As such, a higher level number is taken to be

lower in the tree, and thus at a finer resolution. To avoid confusion, "coarser" and "finer" are used for the remainder of the document to describe relative voxel, or node, levels. We also introduce some notation: V_n^m represents the n^{th} node at level m in the octree, where $m \in [0, L]$ and $n \in [0, 8^m - 1]$.

The cells are handled somewhat differently than voxels: cells are not approximated, only the voxels they contain are. In this sense, cells are like Luebke's faces, and voxels like his vertices. If cells are approximated to varying sizes, cracks and gaps appear in surfaces generated across 2 or more cells of varying level. This problem is discussed in much detail in [MSS94, SFYC96].

To resolve this, HIME stipulates that cells all be of a homogeneous size – a size equivalent to that used by the leaf level voxels. To satisfy this requirement, the notion of a ***voxel proxy*** needs to be introduced: if a cell contains voxels not at the leaf level, these voxels must approximate the leaf voxels required to form the cell; that is, they act as leaf voxel proxies, as indicated by the gray circles in figure 3.1.

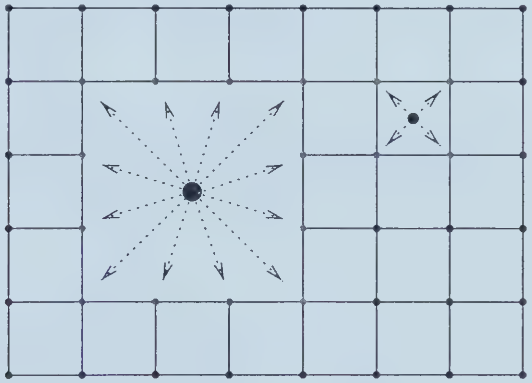


Figure 3.1: Corresponding voxel proxies, represented by gray dots. The solid lines indicate cell boundaries which is why the voxels, indicated by all dots, fall at the corners. The arrows indicate which voxel centre gets propagated to which proxy.

A nice corollary falls from this – cells composed of voxels all represented by the same proxy may be ignored. Only those cells that lie on the boundary between adjacent voxels are of interest, as polygons will only be generated by marching cubes between 2 voxels that transition from "in" to "out". This dramatically reduces the amount of MC cubes needing to be processed each frame. Cells falling inside a voxel space are labelled ***subcells***, like Luebke's subfaces, whereas the cells on the boundary between 2 voxels are simply labelled cells. In figure 3.1 the large voxel on the left encompasses 9 subcells, and the smaller one on the left contains 1 subcell.

With this we also define the notions of an ***active cell list*** or ACL, and an ***active voxel list*** or AVL. The AVL is a linked list of voxels, not necessarily leaves, forming a breadth-wise

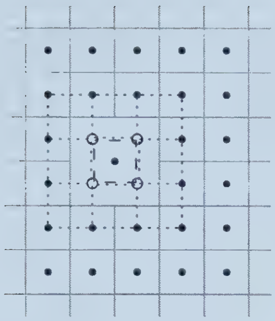
cut through the tree. The ACL is made up of those cells that are not subcells of any voxel in the AVL; it is the ACL that contains the geometry to be rendered each frame. Cells are added or removed from the ACL when they leave or enter subcell status. Cells in the ACL are labelled as "boundary" cells, with all cells higher, or coarser, in the tree being labelled "active" and all those lower, or finer, in the tree labelled "inactive".

Now that we've defined the high-level structures required for surface approximation, we need to describe their associated functions. Cells enter or leave subcell status as voxels are folded (8 children replaced by 1 parent) or unfolded (parent replaced with 8 children). Pseudo-code for these routines is given below.

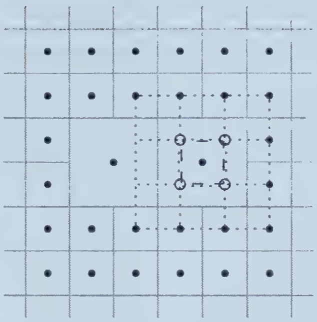
```
fold(voxel V)
     Check if any of V's children are 'active'
          If so, fold each active child
     Set all children to be 'inactive'
     Set V's label to 'boundary'
     //Update proxy pointers for all cells at V's border
     Update V's proxies
     //Update the surface for V
     Update any surfaces contained by cells referencing V
     Remove all V's subcells from the ACL

unfold(voxel V)
     if V has any children
          Set V's label to 'active'
          For all V's children v
               Set v's label to 'boundary'
               //Update proxy pointers for all cells at V's border
               Update v's proxies
          //Update the surface for V
          Update any surfaces contained by cells referencing V
          Add V's subcells to the ACL
     else
          Set V's label to 'boundary' and exit.
```

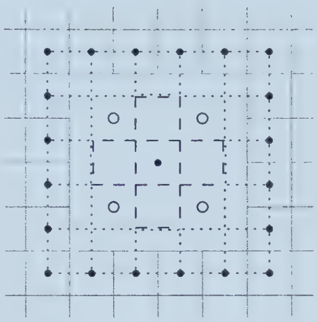
Figure 3.2 depicts a sequence of successive folds and the resulting voxels. The solid black dots represent a given voxel's centre, with the hollow dots indicating the centres of those voxels that were folded during the current operation. The dotted lines indicate the


Folding 4 leaf voxels into their parent

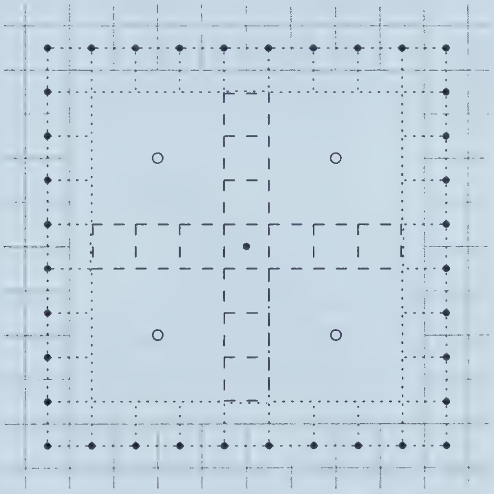
(A)


Folding 4 leaf voxels again

(B)


After folding 2 more sets of leaf voxels, fold the resulting 4 parents

(C)


After repeating the previous fold 3 more times, fold the resulting 4 voxels into their respective parent voxel

(D)

Figure 3.2: Fold sequence

cells which will need to be updated as a result of the operation and the solid gray lines indicate a particular voxel's boundaries. Lastly, the dashed lines indicate the subcells that were removed during each operation.

3.1.2 Detail Metric

To control the folding and unfolding of nodes the notion of a ***detail metric*** needs to be defined. The detail metric is a measure used to decide which nodes should be folded or unfolded each frame. With the addition of a detail metric, the MC algorithm can calculate an accurate representation of the surface without having to traverse the tree all the way to its leaves. Instead of using some constant value to control the depth of traversal, HIME's

system is entirely dynamic, relying on the screenspace extent of a given node to decide if traversal should continue. The screenspace calculation is described further in Appendix A. The system used in HIME is similar to that used in [LE97]. Octree nodes that cover little screen space can be folded into their parent, with the child becoming inactive and the parent being labelled as a boundary node. Conversely, nodes that increase in apparent size are unfolded, with the parent being labelled as active and the children becoming boundary nodes. As such, all nodes above a boundary node are active, and all below are inactive.

The set of nodes contained in the AVL describes the surface, with the displayed polygons being built from these nodes' vertex information and the related ACL. The AVL is initially created from the root node of the tree, and unfolding each successive child until the screenspace measure has been satisfied. For each successive frame the AVL is traversed - the screenspace extent is calculated for each node along the path and a decision is made to fold, unfold, or leave the node alone. If the node is folded/unfolded, the traversal continues with the newly added/deleted nodes.

3.1.3 Simple Marching Cubes

HIME uses a simplified version of the marching cubes algorithm in an attempt to speed rendering and surface generation. The method used is very similar to that seen in [GH91, Mel98]. When polygonizing the surface, a fixed isosurface value is used with no interpolation. The surface is generated with an isovalue of 1.0 (assuming the dataset is clamped to 1.0), meaning the vertices are only generated at the cell corners.

Based on this assumption, a reduced base set of cases can be generated. These were first set out in [Mel98]. As the isosurface threshold approaches 1.0, some of the triangles in the original MC cases collapse to a vertex or an edge. For example with case #30, the initial 4 triangles collapse into a single triangle:

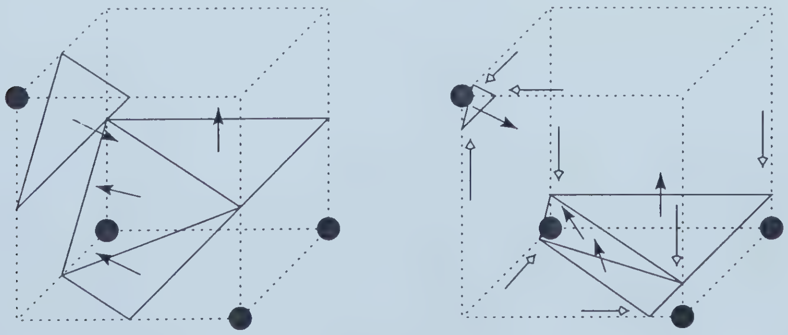


Figure 3.3: MC case #30 collapsing from an insovalue of 0.5 to an isovalue 1.0

We refer to this simplified marching cubes algorithm as ***SMC***, and the original marching cubes as ***MC***. Collapsing the edges for all MC cases results in a set of 12 base SMC cases

which will generate at least one polygon. All other cases may be generated by taking symmetries of these base cases. The entire set is shown in figure 3.4.

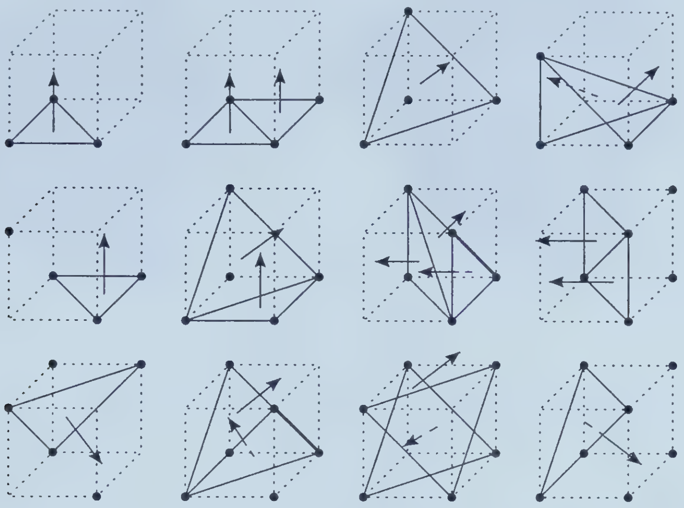


Figure 3.4: SMC base cases

It must be noted that in HIME, the cell corners are located at the voxel centres. Because adjacent boundary voxels can be at varying levels within the octree, it is possible to obtain an irregularly shaped cell with non-orthogonal edges.

SMC produces a slightly cruder model, with a constant reduction in output polygons, speeding rendering. More importantly, since SMC locates vertices at cell corners, it avoids the necessity to reinterpolate face vertex positions that would arise with the use of MC as cells are distorted during voxel folds and unfolds.

3.1.4 Normal Calculation

Normals for all isosurfaces are generated on a per-vertex basis allowing smooth shading of surfaces. Normals are precalculated for each level in the tree, and are updated only during collision. Because they are calculated on a per-vertex basis, they can be stored on the voxel with only a small increase in memory usage. The pre-calculation computes a 26-point gradient across the neighbouring voxel values in the current level for a given voxel, with the collision update function using a 6-point gradient as described in [LC87]. Since surface modification should be as interactive as possible, a cruder normal calculation method is acceptable. Interactivity is not crucial, allowing more accurate calculation. A comparison of visual results is shown in fig. 3.5, with a further discussion of computation time given in chapter 5.

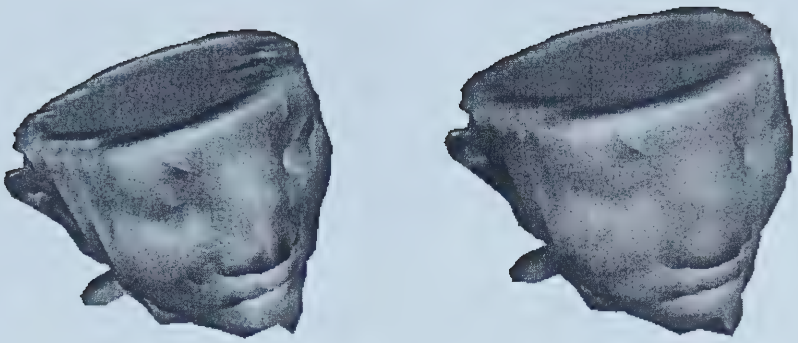


Figure 3.5: 6-point gradient vs. 26-point gradient

3.2 Rendering

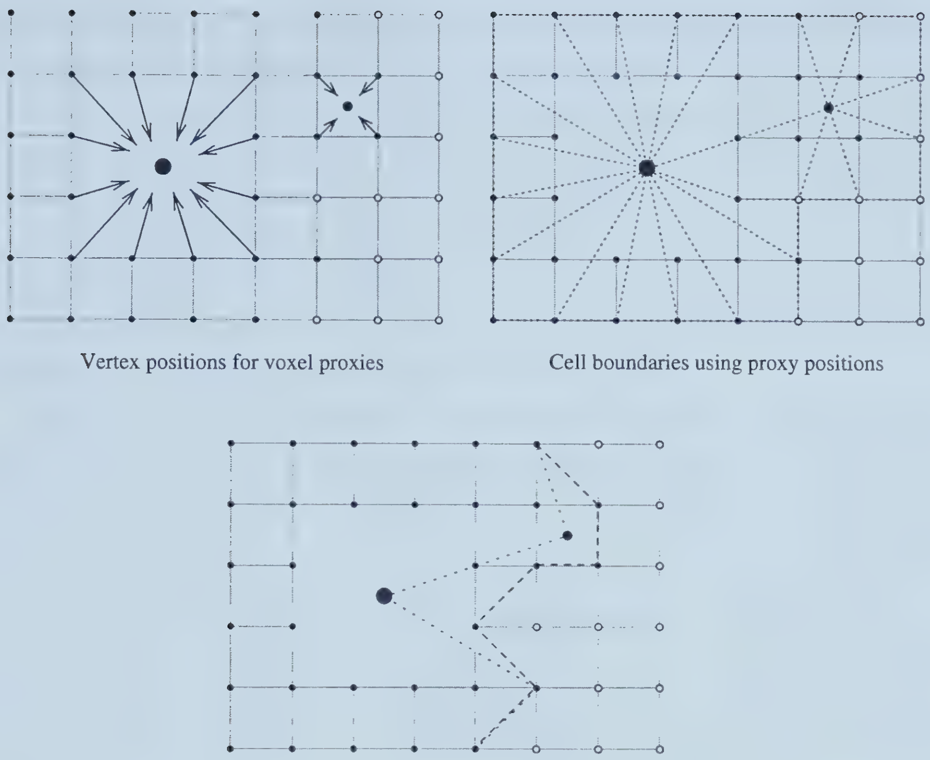
Once the folding and unfolding process has completed for the current frame, polygons need to be generated. Vertices for each cell are generated at the centre of each voxel proxy in accordance with the SMC algorithm. The problem comes when trying to generate a surface across voxels that are of a different size. This was solved with the definition of a voxel proxy.

A marching cubes case is generated in accordance with the smallest cell size regardless of the actual level of the surrounding voxels. The corners are approximated by the given voxel's proxy, which corresponds to the value of that proxy. The vertex positions also correspond to the centre of the proxy, allowing the proxy to contain all pertinent rendering information.

Fig. 3.6 illustrates the functionality of the renderer. Solid black dots are inside the surface, the hollow dots are not. The first image shows the world location of the voxel proxies, with the arrows pointing to where a vertex would be generated if it would fall on that particular voxel.

The dotted line in the next image marks out the actual cell boundaries across which the SMC process will iterate. There is even a degenerate cell within this example - the cell in between the 2 folded voxels which has an edge running across the top of both these voxels becomes a single line. Ultimately the SMC process doesn't care about the actual size of the cells that it processes, this degenerate cell will be treated as though it were like any other cell. This type of cell can actually generate triangles, section 4 discusses how to avoid doing so.

The dashed line in the final image represents the surface as it would be generated by HIME, with the dotted line representing the surface as generated by the classic marching cubes algorithm with an isovalue of 1.0.



Vertex positions for voxel proxies

Cell boundaries using proxy positions

Surface generated by classic MC vs. surface generated by SMC

Figure 3.6: Surface generation. The gray lines indicate cell boundaries in all figures, with the dots indicating voxels: hollow dots represent voxels classified as "out" and solid dots represent voxels classified as "in". The arrows indicate the direction a cell corner will shift as a result of the voxel proxy, with the second figure illustrating the result of that shift: the dotted line indicates the new locations of the cell boundaries. In the third figure, the dashed line indicates the surface as generated by classic MC with the dotted line being that generated by SMC.

3.3 Collision

The collision detection system in HIME builds on the work done in [MPT99, GH91]. The colliding object is represented as a point shell with a granularity greater than that of the voxel surface being collided with. This higher resolution reduces aliasing artifacts as the colliding object intersects the surface, but has the downside of requiring more collision checks.

There are many ways of creating this point shell. The method in [MPT99] creates a volume occupancy map by voxelizing the given object and using the voxel centres as the point shell. A simpler method is to use the vertices of the object as the points of the shell; while not at all optimal, this method is very convenient. It suffers from having a poor

distribution of points across the surface. Planar surfaces will create large gaps in the shell, while areas of high curvature will tend to be quite oversampled.

Regardless of the creation method for the point shell, determining if this shell has intersected the surface of the HIME object is trivial due to the voxel-based nature of the algorithm. Voxel-based volumes are characteristically square or rectangular in shape, giving a perfect fit for a bounding box. If the point shell has a bounding volume of its own, a cheap bounding box test can be made to determine if the shell lies within the voxel space. If it is found to be within the space, each point on the shell is transformed into voxel space, discretized, and then used as a look-up key into the voxel array.

The problem comes when we factor in the hierarchical nature of the HIME voxel structure. Collisions occurring with a node at a given level in the tree need to propagate to the corresponding child and parent nodes to maintain a coherent surface across all levels. It must be noted that HIME first calculates all intersections starting with the leaf level. As such all changes propagate upwards, and downwards propagation is happily ignored.

We adopt a simple binary propagation method: a parent node is updated only if more than half its children are collided with and flip from “in” to “out”. By this, parents that initially have less than half of their children marked as “in” will never be updated as a result of collision propagation. If a parent does receive a collision propagation update, its isovalue is set to zero.

Pseudo-code for the collision algorithm follows:

```
collide()
    if the bounding boxes of the object and volume intersect
        for every sampled point P in the object
            if P is in the volume bounding box
                find the voxel containing P
                if the voxel is inside the isosurface
                    empty the voxel
                    amongst the voxel ancestors
                        empty any voxel with less than 4 filled children
```

Cycling through each point on the shell of the colliding object can be expensive, since each point needs to be multiplied by a transformation matrix. A simple optimization is to check if the shell’s position has changed since the last frame – if it hasn’t, the entire collision detection function may be skipped.

This binary method was used in order to make the collision detection and updating routine as fast as possible. A potential improvement is to re-average the parent’s isovalue

whenever one of its children was modified, giving a smoother surface. Care must be taken to prevent each leaf node modification from propagating to the root of the tree, as re-averaging a given parent will then cause the parent's parent to be updated.

Chapter 4

Optimizations

A naïve implementation of HIME can result in disappointing performance and speed. Fortunately, there are several key elements that can be exploited for some notable gains.

4.1 Avoiding Traversal of Redundant Cells

Early on a decision was made to traverse cells at the leaf level and generate surfaces there. As a reminder, this involves traversal of the boundary voxels and examining the cells they proxy. This is done as a solution to the cracking problem that would arise if the cells were differently sized.

The problem with this traversal of the leaf cells is that when rendering high-level, or very coarse, boundary paths an enormous number of cells are visited for a relatively small number of voxels along the boundary path, even when the subcells are skipped. As a direct result, the polygon count for a highly approximated model is uncharacteristically large. This occurs because at high levels in the tree the voxels become quite large in size relative to the leaf cell size. When several voxels of a similar level are adjacent there are a large number of cells that share the same proxies, and thus will create the same triangles - in most cases these triangles will be degenerate, as previously shown in figure 3.6.

Figure 4.1 illustrates a 2-D situation from which this redundant triangle creation, or overdraw, will arise. The cells outlined with a dotted line are those that will create overdraw and cells with a dashed line will not cause overdraw. V_1^{l-2}, V_2^{l-2}, V_3^{l-2} and V_4^{l-3} are voxels whose centres serve as proxies for the surrounding voxels. Subcells for these voxels have been removed.

This overdraw problem occurs when updating the surface for cells surrounding a given voxel, V_2^{l-2}. A particular cell on the surface of V_2^{l-2} will be composed of proxies from adjacent voxels, in addition to a proxy for V_2^{l-2}. Overdraw occurs when a triangle is created with 2 or more vertices falling on corners sharing the same proxy - for example cell C_1 in figure 4.1 will always generate degenerate triangles whereas cell C_4 may or may not; the

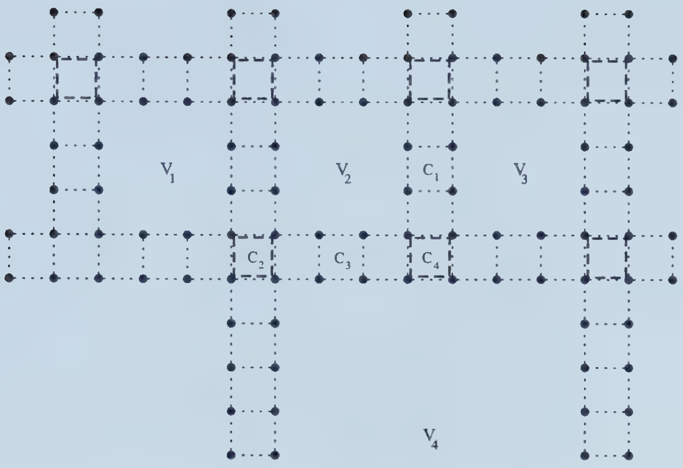


Figure 4.1: Example illustrating the different cases arising from the overdraw problem and its solution

upper and lower cell neighbours of C_1 will generate exactly the same tris as C_1. In the case of C_4, any triangle with any 2 vertices falling on the bottom 2 corners will be degenerate, while triangles with only 1 vertex falling on either of these corners will still be valid.

4.1.1 Level Test

We handle overdraw with a recursive, voxel face based algorithm. If we're updating voxel V_x^m and we need to calculate the surface at a shared face between voxel V_x^m and voxel V_y^n 2 situations may arise:

- ***m* coarser than *n*** In this case there will be at least 3 other voxels sharing portions of V_x^m's face in addition to V_y^n. Generate surfaces for the corner cells of V_x^m's face, then subdivide and recurse.
- ***m* finer than or equal to *n*** The face is shared, or only covers a portion of V_y^n's face, with the remainder being shared with at least 3 other voxels. Generate surfaces for only the corner cells of V_x^m's face.

Only the first case in this test requires recursion – the last case results in no additional triangles being generated other than at the 8 corner cells of the voxel. Pseudo-code for a surface update function implementing this test is given below, with a detailed explanation following immediately after. The code refers to figure 4.2.

```
UpdateSurface(voxel)
    generate surface triangles in corner cells
        if voxel is not a leaf
```

```
            for each voxel edge
                UpdateEdge(voxel level, edge)
            for each voxel face
                UpdateFace(voxel level, edge)

UpdateEdge(voxel level, edge)
    if level of at least one voxel in edge center cell is finer than voxel level
        generate surface triangles in edge center cell
            if voxel level is not leaf-1 level
                UpdateEdge(half 1 of edge, voxel level+1)
                UpdateEdge(half 2 of edge, voxel level+1)

UpdateFace(voxel level, face)
    if level of at least one voxel in face center cell is finer than voxel level
        generate surface triangles in face center cell
        if voxel level is not leaf-1 level
            for each face quadrant
                UpdateFace(voxel level+1, quadrant)
            for each interior face quadrant edge
                UpdateEdge(voxel level+1, quadrant edge)
```

We begin with a discussion of updateSurface(). The corner cells of a voxel are updated regardless of the levels of the adjacent voxels, as there are no other cells along any of the edges that will share their same 8 voxels. This is indicated in figure 4.2(A): the black circles indicate voxels proxied by V, with the sparse dotted line indicating the boundary of V. The gray circles indicate the proxies belonging to neighbouring voxels. A related example can be seen by observing cell C_2 in figure 4.1.

An optimization is made for the voxels at the bottom 2 levels in the tree, and is used to end recursion. A leaf voxel is covered by a 2^3 cell cluster, and so processing the corner cells for a voxel at this level is sufficient. An $L-1$ level voxel is covered by a 3^3 cluster; processing the corner cells, the edge midpoints, and the face midpoints is enough to update the surface for a voxel of this level.

On to updateEdge(): for voxels not belonging to either of the 2 finest levels, we need to subdivide. First, the 12 edges for V are subdivided in turn and a check is performed at the cell straddling each edge midpoint. The cells are processed per-edge because each cell is a part of 2 faces, and processing it on a per-face basis results in duplication of work.

Since a voxel will always number an even power of 2 proxies along its edge, there will always be an odd number of cells covering that same edge guaranteeing a cell across the

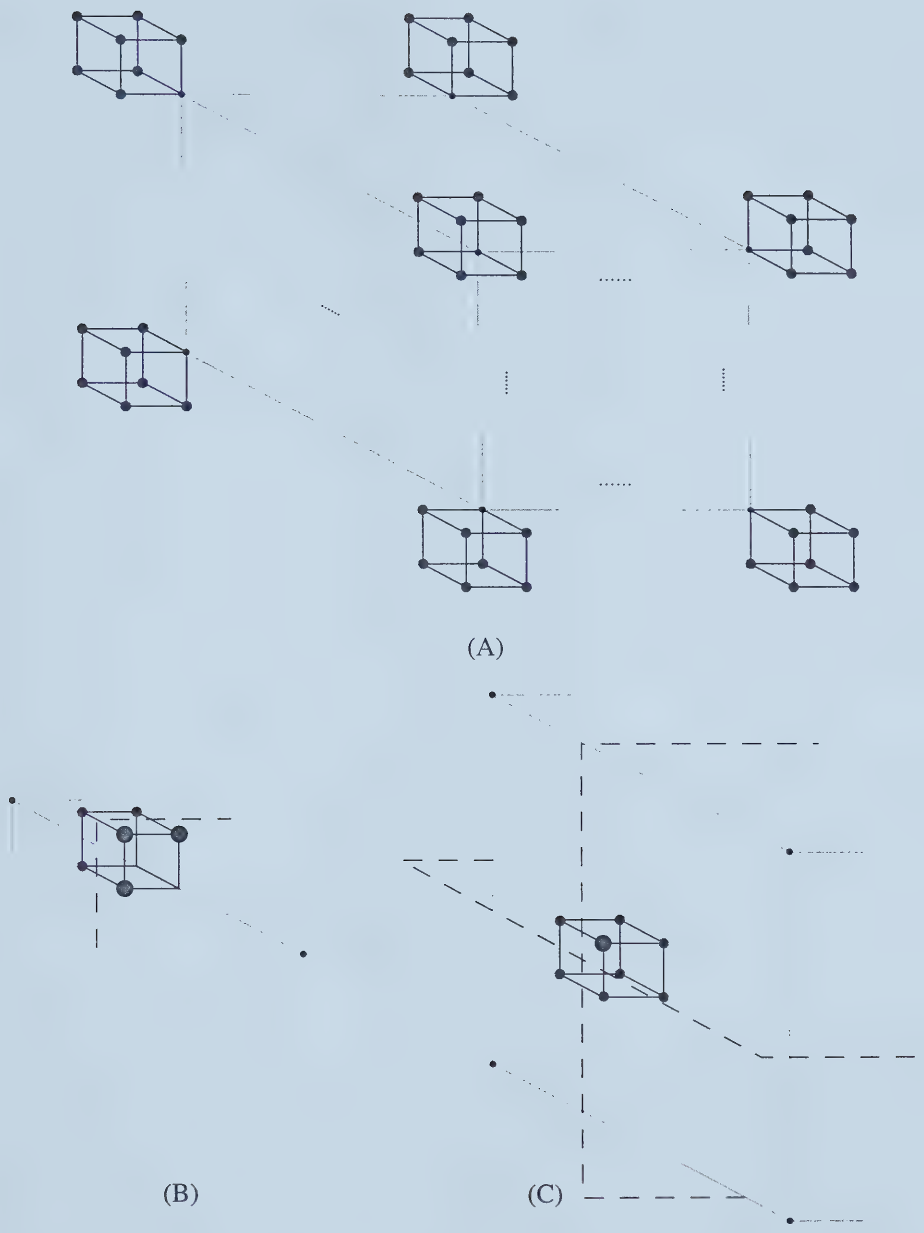


Figure 4.2: Surface update example illustrating the different 3-D cases of the solution to the overdraw problem. Dots with a black centre indicate voxel proxies belonging to the voxel being currently updated.

midpoint. The only case for which this is not true is the leaf level, but as mentioned earlier the leaf level is treated as a special case.

The mid-point cell of an edge is adjacent to a potential of 6 other voxels, as seen in 4.2(B). The dashed line indicates one of the 3 planes along which any of the adjacent voxels may be

split. Its voxel proxies, represented by gray circles, are checked for their level in the octree. If they have the same or coarser level than the current voxel, the proxy values indicated will be the same for the entire length of the edge resulting in the same SMC case being generated and thus the same (degenerate, it turns out) triangles being created – otherwise, we subdivide and recurse. Note that we need to check only 3 of the 6 voxels for their levels, since same or coarser level voxels will not change across an edge. The proxies to be tested are indicated in figure 4.2 by the outlined gray circles.

Faces are handled similarly in updateFace(), except that instead of performing 3 neighbouring proxy checks we can get away with performing a single check. Figure 4.2(C) demonstrates this, with the outlined gray proxy being the neighbouring proxy selected for testing.

Another difference between edge subdivision and face subdivision is that subdividing a face clearly results in 4 quadrants instead of 2 half edges; the quadrants are processed recursively. The edges of these new quadrants falling on the inside of the face need to be passed down to the edge update function, with the remainder of the cells passed to the face update function. These edges are made up of cells straddling the subdivision boundary - in figure 4.2(C) these cells would correspond to the cells straddling the dashed line, which in the figure have been omitted for clarity.

We should note that this splitting algorithm is made possible by a nice corollary of the BONO algorithm. When subdividing a volume, BONO creates octants that are always an even power of 2 in size along an edge. Thus, when a volume is split into new octants, the boundaries of these new voxels intersect the edges and faces of the original voxel exactly at the midpoints. Since the surface is generated along these voxel boundaries a simple level test, described earlier, can be performed at these midpoints to determine how best to polygonize a given boundary.

Another side effect of the BONO splitting mechanism is that adjacent voxels will never be staggered - if the level of voxel V_x is coarser than or equivalent to that of it's neighbour V_y, the corners of V_x will never fall on an edge or face of voxel V_y. Referring back to figure 4.1, voxels V_2^{l-2} and V_3^{l-2} will always have their corners and centres line up: a corner of V_3^{l-2} will never fall on a face of V_2^{l-2}. In addition, the outside corners of V_3^{l-2} and V_4^{l-3} will always line up, with a corner of V_3^{l-2} always falling on the centre of one of V_4^{l-3}'s faces.

A simple 2D example can be found by referring back to figure 4.1: when updating V_2^{l-2}, cells C_1 and C_3 will fail the level test as the levels on either side are equal or coarser than the level of V_2^{l-2} - no triangles will be generated, and no subdivision will occur. Cell C_4 will not be subdivided because it falls on the corner. If V_4^{l-3} is the voxel being updated however, cell C_4 will be processed because it falls in the middle of V_4^{l-3}'s edge. The edge will then be subdivided with C_3 being the next cell tested. Again it will fail because the levels of the neighbouring proxies will be equal, and the algorithm will move on to the remaining half of

V_4^{l-3}'s edge.

4.1.2 Performance Numbers

All testing done for these optimizations was done on goodwin, except where noted. Tests were done with 2 datasets: a 41^3 procedurally generated sphere, and a $64 * 64 * 93$ CT study of a male head. The testing procedure involved rotating the given dataset at a constant speed for 5 seconds, with samples taken at the end of each frame. The 5 seconds are started as soon as the first frame is rendered, allowing the system to initialize and find equilibrium. As such, there is a delay in rendering the first frame since the system unfolds from the top of the tree to obtain the boundary path; this initial frame is not included in the plots as it is 1-2 orders of magnitude larger than any of the other samples.

The first set of graphs indicate the performance improvement in the number of updateSurface() calls before and after implementing the redundant cell avoidance optimization. The gray line is the initial implementation, with the black line being the optimized version; the graph on the left corresponds to the sphere dataset, on the right the head dataset. To make the graphs more readable, frames making no updateSurface() calls are ignored, resulting in gaps in the graph lines. The performance improvement shown by this

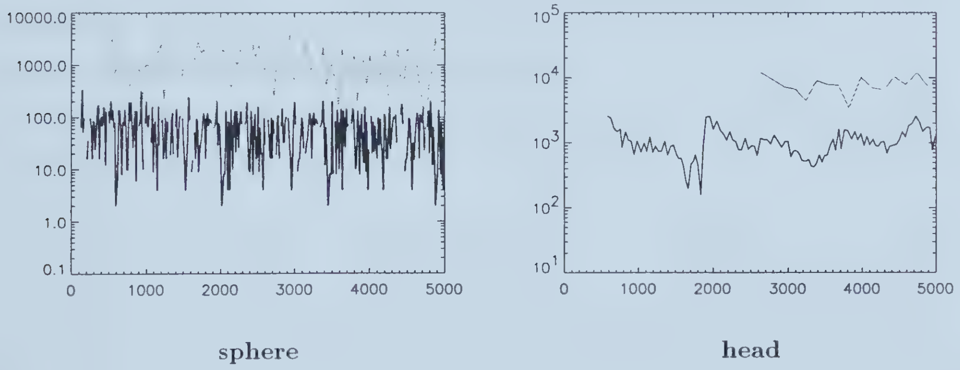


Figure 4.3: updateSurface() calls vs. total time (ms), before and after implementing the optimization

optimization is quite remarkable – a gain ranging from one to two orders of magnitude is shown in figure 4.3. This clearly has an effect on the frame rate, shown in the increase in frequency of samples taken for the head dataset, and decreased initialization time for the first frame. The effect of this optimization on polygon count is also substantial, shown in figure 4.4.

The gray lines again indicate the original, the black the optimized version; the solid line indicates the total polygon count for the model whereas the dashed line represents the number of degenerate triangles within that total. The dashed line is difficult to make out

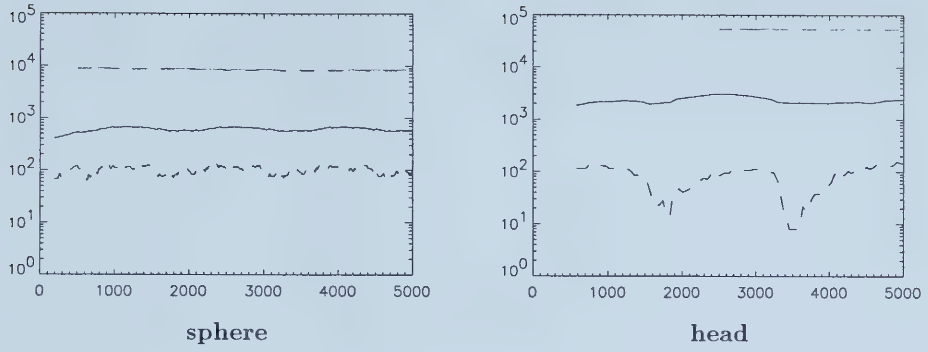


Figure 4.4: Total triangles(solid) & degenerate(dashed) triangles vs. total time (ms), before and after the optimization

for the original implementation, indicating just how much of the polygon count is made up of degenerate triangles.

4.2 Avoiding Generation of Degenerate Triangles

The aforementioned optimization performs quite well in reducing the number of pointless updateSurface() calls per frame; unfortunately it does not completely do away with degenerate triangles. A problem arises when running SMC on the cells selected by the recursive subdivision algorithm - a basic implementation of SMC doesn't take into account the fact that certain cell corners can be represented by the same proxy. This was alluded to in the description of figure 4.1, where cell C_4 may or may not create degenerate triangles depending on the orientation of the triangle.

A further optimization can be made by taking further advantage of the notion of proxies. Referring back to figure 4.2 and the face update, we can make some assumptions with regards to polygonizing the face's centre cell. The inside 4 voxel proxies of the centre cell, represented by black dots, will always be represented by the same proxy value. Thus, any triangle's vertex falling on any of those 4 corners will be represented by a vertex at the centre of the current voxel. If 2 or more vertices belonging to that triangle fall on any 2 of these 4 corners, the triangle will be degenerate.

This problem crops up while performing the edge update as well, but instead of having only 4 proxies represented by a single proxy value at least 2 will be shared with a potential of having up to 6 shared proxies, not necessarily all by the same voxel. Recall that when performing the level test on the edge midpoint cell, we process the 3 neighbouring proxies until we come across one that is finer than the current voxel, at which point the algorithm stops and subdivides the edge. The remaining proxies can be at any level in the tree, which can cause other edges or even faces to be represented by a single proxy.

To deal with this we add an extra parameter to the polygonization call that indicates which corners share the same proxy. When creating triangles a simple check is done for each triangle to determine if 2 or more vertices fall one one of these shared-proxy corners - if so the triangle is not added to the render list.

4.2.1 Performance

We use the same testing procedure to evaluate this optimization, the results of which are shown in figure 4.5. The number of degenerate triangles rendered per frame are tracked with the comparison made between the first optimization, represented by the gray line, and the combination of the first and second optimizations represented by the black line. Unfortunately, the sphere dataset gives almost imperceptible results and is not included. The results from the head dataset are more noticeable, but are nowhere near the scale of the improvement shown by the first optimization.

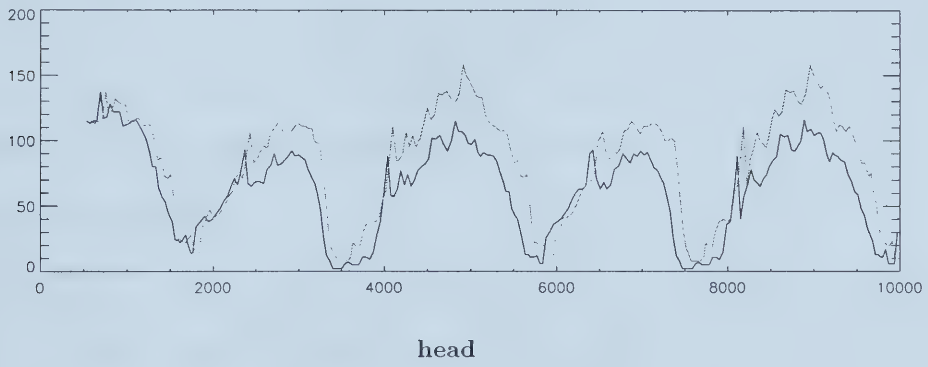


Figure 4.5: Degenerate triangles vs. total time (ms) before and after implementing the SMC optimization

Clearly, the limited optimization we perform here shows its weakness. By focusing only on degeneracies arising within the current voxel, we cannot eliminate the majority of the degeneracies. Full elimination would require shared vertex checks in neighbouring voxels as well. Since HIME is not render bound, we decided against increasing the complexity of updateSurface() to reduce triangle count.

4.3 Performing Surface Generation Only On Boundary Cells

Another simple optimization ensures surfaces are generated only when the boundary is well-known. Recall that the fold/unfold process runs first to update the boundary path nodes, and then the renderer makes a second pass to display the new triangles. The fold/unfold process can traverse multiple levels of the octree in the course of a single frame. During

an unfold, the parent node is removed from the boundary path and its children are added. We could update the children immediately. However, this expensive work would be wasted if these children were also unfolded during the same frame, which can happen during large viewpoint shifts.

A better way of dealing with this might be to update surfaces after the boundary is updated, during rendering. The fold/unfold process would add all necessary nodes to the boundary path and then rather than updating the surface immediately, flag them as needing a surface update. As the renderer traverses the boundary path, it then updates the surface for any marked nodes.

This same concept can be applied to collisions. Currently, when a collision occurs the value for the intersected voxels is set to zero, affecting also their proxies and necessitating a surface update for all adjacent cells. Since these same voxels may not even be on the boundary path, it is senseless to allocate processor time immediately to generate triangles for these nodes. Marking them guarantees that when they do become part of the visible surface, the renderer will update them accordingly. The only difference between the collision update and the general update is that after a collision, the normal and interpolated position (see section 4.5) for a given vertex will need to be recalculated.

4.3.1 Performance

The results shown in figure 4.6 are the updateSurface() calls for the head dataset: the gray line indicates the performance given by the combination of the first 2 optimizations, the black line shows the performance after the addition of the marked cells.

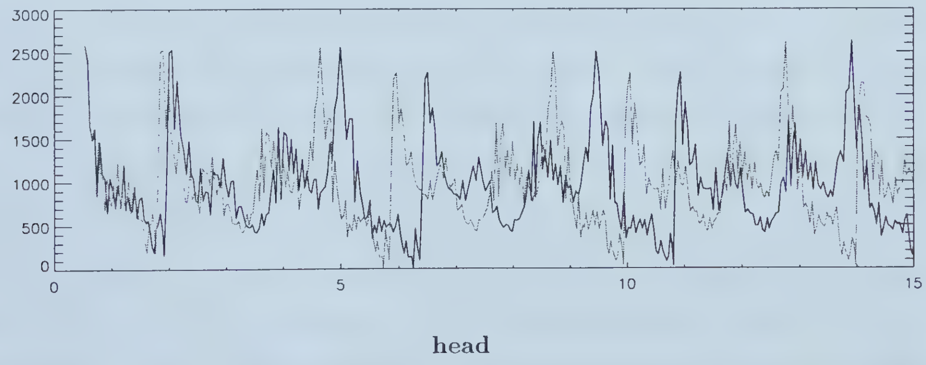


Figure 4.6: updateSurface() calls vs. total time (s) before and after implementing the dirty bit

An interesting phenomenon is observed – not only does this optimization do little to decrease the number of updateSurface() calls, it also serves to increase the frame time, causing the black line to shift to the right. The increase in frame time can be attributed to a reduction in cache coherence when implementing this function. The nodes along the

boundary path need to be swept through to be flagged as needing an update or not. A second pass then needs to be made to update all appropriately flagged nodes; the majority of the nodes will have been flushed from the cache during the first pass, and will need to be reloaded during the second.

The lack of a change in the number of updateSurface() calls may be explained by very few nodes along the boundary path needing to be folded or unfolded more than once in the course of a frame. This may be a product of the test session, which rotates the model at a constant speed about it's centre: a given node may not move enough in the course of a frame to require folding/unfolding multiple times. Another test examining this would be to track the surface updates while zooming in on the model as it rotated. Results of this second test are shown in figure 4.7.

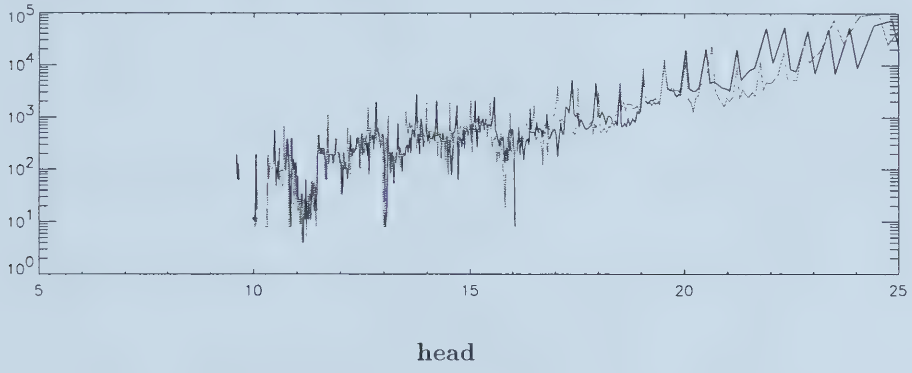
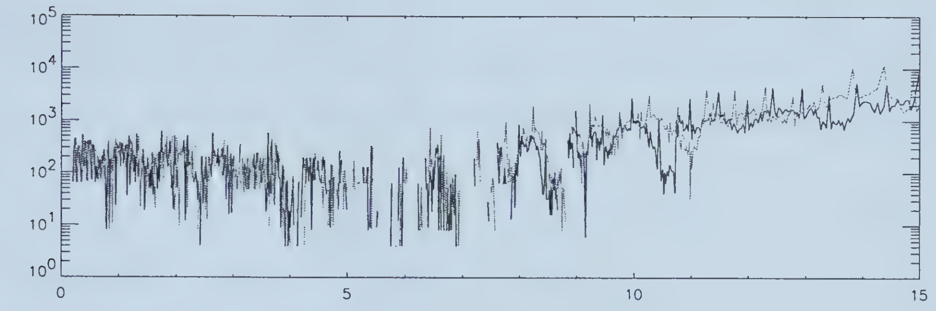


Figure 4.7: updateSurface() calls vs. total time (s) during a session in which the **head** dataset is gradually brought closer to the viewpoint

Again, little improvement is shown. The large gap at the beginning of the graph is the result of the viewpoint being a great distance from the model – the model is simply not close enough to require nodes to be unfolded. In the course of 15 seconds it is possible to go from a very coarse approximation of the model all the way to the leaf representation, suggesting that the model is too small to register any gain from this optimization. We perform the same test on a $128 * 128 * 93$ version of the head dataset, with the results shown in 4.8.

No discernible improvement is shown. Figure 4.9 illustrates the difference in updateSurface() calls between the non-cell-marking version of HIME and the cell-marking version. A large number of positive results in that figure would have demonstrated that the cell-marking version indeed made a difference with the non-marking version consistently making more updateSurface() calls. This is not the case, leading us to believe that this optimization is ill-suited for HIME, especially when objects are viewed from a distance.



$128 * 128 * 93$ head

Figure 4.8: updateSurface() calls vs. total time (s) of a session in which a $128 * 128 * 93$ CT head study is gradually brought closer to the viewpoint

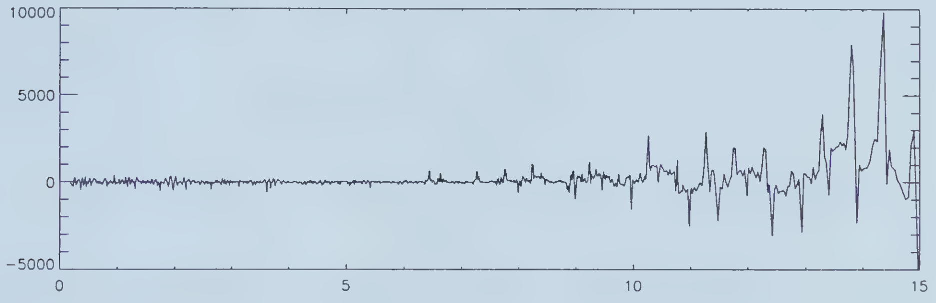


Figure 4.9: Difference in updateSurface() calls vs. total time (s)

4.4 Limiting Surface Change To Control Speed

Luebke describes a method by which a queue is used to limit the complexity of the simplified scene, putting a cap on the rendering time [LE97].

The system used in HIME is quite similar to that employed by Luebke: the screenspace extent for all current boundary nodes is calculated and ordered with greatest extent first. The list of active nodes is in practice quite similar from frame to frame, meaning there should be only a small number of nodes needing to be folded/unfolded throughout the course of a single frame. Those nodes marked for folding are folded immediately while those marked for unfolding are placed into a separate Unfold list. Folding is immediate because it removes triangles and nodes from the boundary path which decreases rendering time - the more folded nodes during the course of a frame the better. Once the extent calculation and folding has completed, the Unfold list is sorted from greatest to least projected size and is traversed in that order, unfolding nodes along the way until the maximum number of unfolds for a single frame has been reached. A nice corollary of this system is that the user has complete control over visual fidelity: if the user stops interacting with the model

they can watch as the model slowly refines to a greater level of visual accuracy. If the user is constantly rotating the model, the Unfold list changes dynamically and maintains interactivity with only the most visually important nodes percolating to the top and being unfolded.

A key difference between Luebke's & HIME's queueing methods is that Luebke uses triangle count to limit unfolding, while HIME counts unfolds. Luebke's method is that of a ***triangle budget***, meaning that the system can render a maximum of N triangles: unfold the top node in the sorted queue until the number of triangles in the scene has met the budget. In HIME the bottleneck is the updateSurface() function that is called once a node has been folded or unfolded, and not the polygon count. Thus, minimizing the number of updateSurface() calls has a greater effect on performance than minimizing the number of triangles. It is also easier to maintain a constant frame rate when minimizing the number of unfolds taking place - when trying to acheive a certain number of triangles, it is not known how many nodes will have to be unfolded before the triangle budget is met.

4.5 Improving Visual Quality with Vertex Interpolation

Vertex interpolation is not a speed optimization, but rather an optimization to improve the visual quality of the approximation. In [LC87], Lorenson and Cline linearly interpolate the face vertices in a cell between 2 voxels to lie on the cell edge as the algorithm marches through the voxel space. HIME takes a similar approach, but instead of interpolating with a single degree of freedom parallel to the current edge, HIME interpolates with 3 degrees of freedom along the 3 cardinal axes per voxel or face vertex. In HIME, interpolation is

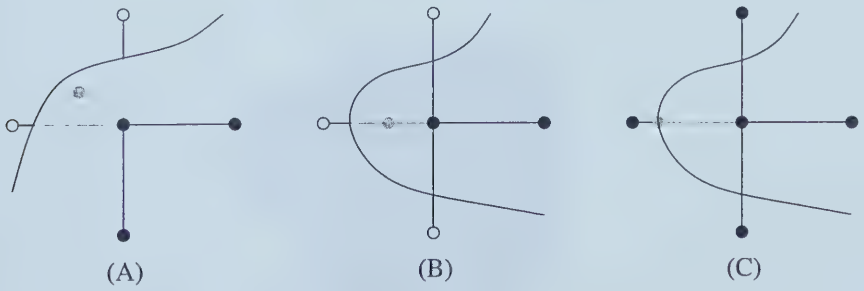


Figure 4.10: Vertex interpolation scheme. (B) and (C) compare before and after the implementation of an additional accuracy optimization

done as a preprocessing step independent of the polygonization phase. Interpolation is done within a given level in the octree: with leaf voxels, then at level $L - 1$, and so on. This is done to remove any sort of inter-level dependancy for the interpolation process to resolve. Interpolation must be recalculated after a collision for leaf voxels and all affected ancestors

and neighbours. For a given voxel, an offset value is determined for each of the 3 cardinal axes. When interpolating along a given axis we check the 2 neighbouring voxels on either side of the current voxel: if both neighbours are "in" or "out" of the surface the offset value for that axis is 0, and we move to the next axis. If only one is "out", we interpolate between that neighbour to determine the offset. Figure 4.10(A) demonstrates the offset calculation. The dashed lines indicate the offset value for each axis – the interpolated point is placed at their intersection. The value is simply half the distance to the surface. Using half the distance to the surface as opposed to the entire distance results in a surface that is less blocky, as shown in figs. 4.12 and 4.13.

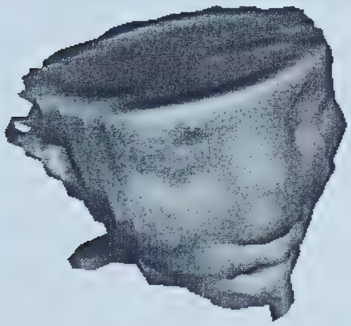


Figure 4.11: Original model

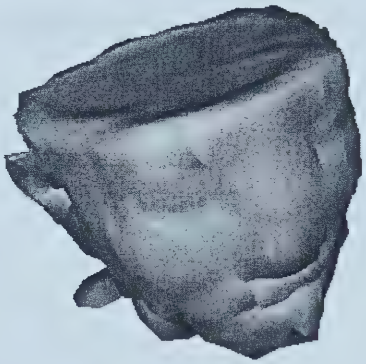


Figure 4.12: Full-distance offsets, for the purposes of comparison

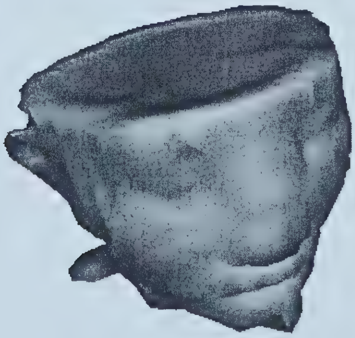


Figure 4.13: Half-distance offsets with an additional optimization. This is what is currently implemented in HIME.

A simple optimization can be made for neighbours having the same polarity, shown in figure 4.10(B). No interpolation will be done along the y axis of the figure, allowing us to simply place the vertex directly on the intersection point of the surface with the x axis. Generalizing this to 3 dimensions, if 2 axes have neighbours with the same polarity, not necessarily the same between the axes, this optimization can be applied.

A comparison between a non-interpolated surface and an interpolated surface is made in figures 4.11 - 4.13; the half-distance offset model in figure 4.13 includes the similar-polarity optimization. The datset used is a $64 * 64 * 93$ CT study of a head, and is rendered at level 3 of the octree which corresponds to a $16 * 16 * 23$ volume. Note that the silhouette of the half-distance interpolated model is much smoother than that of the others – the full-distance interpolated model appears quite swollen and all along the silhouette there are polygons that jut out.

Chapter 5

Testing

Overall performance of HIME, once certain key optimizations have been made, is quite good. HIME is able to acheive interactive frame rates on large datasets with good visual fidelity. The goal of being able to edit the surface in real-time was also acheived. We present an evaluation of HIME's performance in processing and rendering several datasets. The datasets used in this evaluation are given below; the label associated with each dataset is listed, and is followed by a description of the information contained by that particular dataset.

- **sphere41** A $41 * 41 * 41$ procedurally generated sphere. The isosurfaces contained in this dataset represent the distance from the surface to the centre of the sphere.
- **sphere128** A $128 * 128 * 128$ procedurally generated sphere.
- **quarter** A $64 * 64 * 93$ CT study of the head of a 12-year old male. A form of this same set is used in [LC87].
- **half** A $128 * 128 * 93$ version of the head dataset.
- **full** A $256 * 256 * 93$ version of the head dataset.

5.1 Preprocessing

Preprocessing steps are often taken by algorithms to reduce the amount of work done when the algorithm itself runs; HIME is no different. There are 3 main steps to HIME's preprocess:

- **Octree Creation** The volume is subdivided and stored in the octree. Each node in the tree is a pointer to a structure containing the appropriate voxel information. Proxies for the entire tree are also calculated at this time: the given dataset is parsed prior to tree creation, and when a new leaf node is created its proxy is set to the corresponding isovalue from the dataset. Once the entire tree has been built, the leaf

values are propagated upwards to create the proxy values for the remainder of the tree.

- **Vertex Interpolation** The vertices that are rendered every frame are simply the centres of all boundary voxels. As discussed in chapter 4, the visual accuracy of the interpolation can be improved by interpolating the vertices to lie closer to the surface.

- **Normal Generation** Normals for each node in the tree are derived with the knowledge that at the surface of interest between volumes of different densities, the gradient vector is non-zero [LC87]. As a reminder, normals in HIME are calculated with a 26-point interpolation method using the surrounding neighbours for a given voxel.

Before any results are presented, it must be noted that the majority of optimizing was done to improve the rendering performance of HIME. Much work could be done to reduce the preprocessing time as well as the memory footprint, and these are discussed in turn.

The times for each of the preprocessing steps are given in table 5.1. Table 5.1 details the times for each step, given in seconds; the upper half of the table was generated using goodwin, while those on the lower half were generated on cadomin. The times for the **half** dataset on goodwin are skewed upwards due to severe memory thrashing: note that for the **quarter** dataset cadomin is between 20% to 60% faster, while the **half** dataset runs roughly an order of magnitude faster on cadomin. The times for the **full** dataset suffered from memory thrashing as well. The times under the Octree Creation heading detail the time to subdivide the volume, followed by the time to propagate the voxel data from the leaves up to the remainder of the nodes.

Dataset	Octree Creation			Vertex Interp.	Normal Avg.	
	split	fill	total		6 pt.	26 pt.
sphere41	0.26	0.05	0.31	0.54	0.51	2.55
quarter	1.36	0.27	1.63	4.63	3.50	15.85
half	14.48	33.56	48.04	290.56	247.31	333.95
quarter	1.05	0.20	1.25	2.82	2.81	13.07
half	4.45	1.20	5.65	12.22	12.25	55.38
full	31.68	22.15	53.83	182.98	291.57	507.85

Table 5.1: Preprocessing times (sec.)

Averaging the normals for the tree is clearly the bottleneck. The interpolation method used in this process calculates a normal for each voxel within a given level in sequence; such a method would benefit greatly from a caching scheme, especially considering each interpolation requires looking up the 26 nearest neighbours for a voxel. Such a caching scheme is discussed in [LC87, WW92] with respect to the marching cubes algorithm.

Table 5.2 lists the memory footprints for each dataset on the 2 test machines in megabytes, as well as the octree sizes for each volume. The memory usage was determined with

gr_osview [SGI98] and verified with *gmemusage*, an application from SGI's Developer ToolBox which is available by registering as a developer with SGI. Both of these programs have caveats when it comes to displaying exact usage numbers: *gr_osview* will sometimes count memory pages allocated to a running process as being occupied by file system pages; *gmemusage* pro-rates the memory size of a process with the number of processes using each page. As a result, the memory usage as reported by both of these programs tends to be underestimated. The octree size is given in terms of total nodes stored in the tree. The leaf nodes, while stored in the tree, are enumerated separately to illustrate the number of additional nodes created by the BONO algorithm.

Data	Node Count	Leaf Node Count	Memory (MB) goodwin	Memory (MB) cadomin
sphere41	10,840	68,921	10.94	–
quarter	55,151	380,928	43.22	43.27
half	220,601	1,523,712	69.30	164.56
full	882,405	6,094,848	–	509.92

Table 5.2: Additional preprocessing data

It is interesting to note that when viewing the **half** dataset on goodwin, HIME begins to thrash after only occupying 70 MB of the total 128 MB of RAM. It turns out that the IRIX kernel occupies roughly 21 MB, with various daemons and the window manager occupying around 30 MB. As an aside, the kernel on cadomin occupies between 60-70 MB of the 768 available MB, with the window manager and daemons occupying another 70 MB. This indicates the usefulness of a stripped-down system when performing memory-intensive tasks.

As mentioned earlier, minimal memory optimization has been done to HIME. We see that the memory footprint of a given dataset in roughly 2 orders of magnitude greater than the number of values contained in the original set. If we include the leaf nodes in the total node count for an octree, the node count can be quite substantial. As such, minimizing the size of the node data structure would have a noticeable effect on the memory footprint.

Currently, the size of the octree node structure is 68 bytes; for the **full** dataset this results in an octree footprint over over 450 MB. Considering that, at 2 bytes per sample, the original data occupies 11 MB the octree size could certainly be improved upon. The members in the octree node structure are used for tracking folds and unfolds as well as maintaining pointers to any children. It is clear that a large majority of the octree consists of leaf nodes. Fortunately, this can be exploited to reduce HIME's memory footprint: because leaf nodes have no children and will only be folded or unfolded into by their parent, many of the octree node structure's members have no practical use for these nodes. If a simplified structure could be used for the leaves, a large memory savings can be had with no loss in functionality.

A careful examination of the node structure has determined that a potential of 25 bytes can be removed for leaf nodes leading to a projected savings of 145 MB for the **full** dataset, a reduction of 32%. This reduction could be made without affecting the overall speed of HIME – further reductions could be made at the cost of a decreased frame rate.

5.2 HIME

Comparing HIME to any current system is difficult, as there are few systems that provide all the functionality that HIME does. We simply present a detailed analysis of HIME in operation.

Figure 5.1 depicts the view volume as well as the resulting surface approximation with a screen extent of 2.5% once the folding and unfolding is complete.

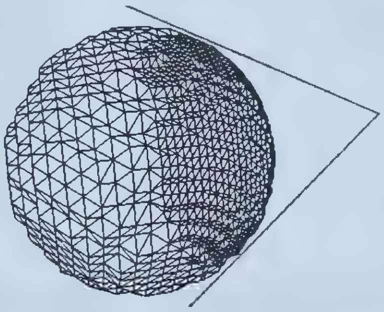


Figure 5.1: Top-down view of **sphere128** showing the view volume. The model was rendered with a screen extent of 2.5%

Figure 5.2 gives the results of rendering the **half** dataset from various viewpoints. Each row of images represents the model as close to a given level in the octree as possible. The **half** dataset creates a tree with 8 levels: the first row of images is the representation stored at the 5th level, the second row the 6th, and the bottom row the 7th. Note that these levels are rough estimates, corresponding to the level value of the triangles composing the majority of those viewable. By carefully observing the wireframe in the bottom row we see triangles belonging to the 6th level along the jawline, and the edge of the right eye socket.

The approximations generated by HIME are quite good – even at the 5th level many of the skull's features are clearly visible, from the upper spine, to the eye sockets, to the jawline. As the resolution of the approximation increases, individual vertebrae and even teeth become apparent.

Figure 5.3 illustrates a shovel bucket as an example of a dynamic object that could be used as the dynamic object in the collision detection. The initial bucket contains 282 unique vertices which are poorly distributed across the surface. Using simply the bucket's vertices for the point shell will result in poor collision results, as the large gaps between many of

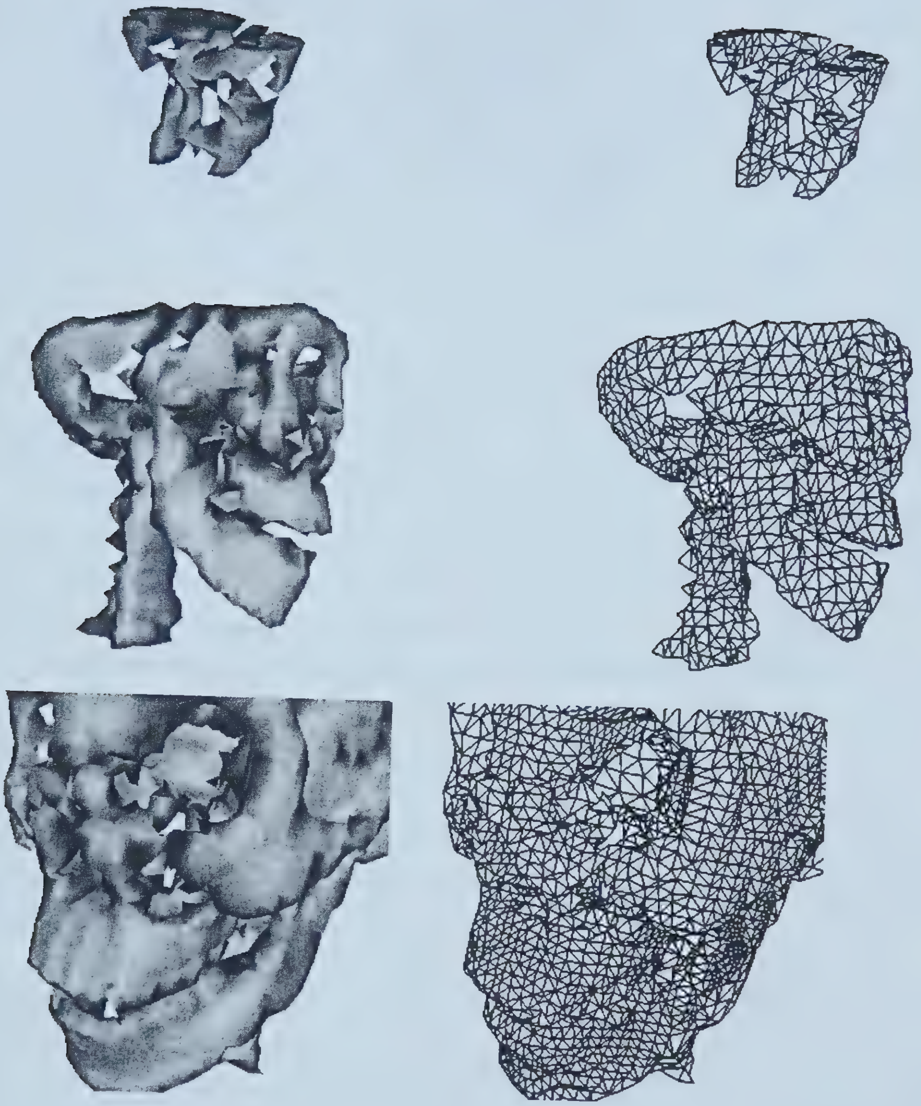


Figure 5.2: Sequence of viewpoint shifts around the **half** dataset

the vertices will allow much of the static surface to pass through. To resolve this, the point shell shown on the far right was constructed by hand and is made up of 110 vertices. This point shell could probably improved upon, as no vertices are located along the shovel sides on as the assumption was made that the shovel would only move as would a real shovel: in a scooping manner. The dynamic object used during testing was a sphere with a point shell containing 812 points evenly distributed across the sphere's surface.

Figure 5.4 illustrates a sequence of collisions and viewpoint shifts with the **half** dataset.

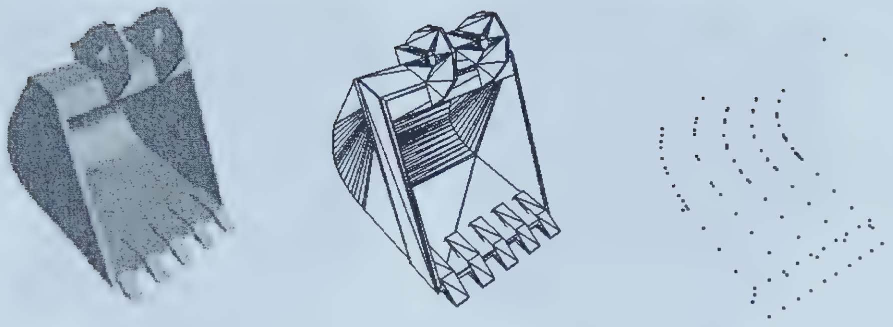


Figure 5.3: Shovel and associated point shell

The first row of figures indicates the original model prior to any collisions.

The second row depicts the results of judiciously applying the sphere to the nose area of the model.

It is clear that the 6-point normal calculation used to regenerate the normals is inadequate in its preservation of surface continuity. At first glance the normals appear to simply be reversed, but by observing the small lumps left floating in the hole created by the sphere we see that the normals are correct for those pieces.

Part of the problem with the normal regeneration lies with the way the collision detection algorithm updates the surface: any voxel it intersects has its proxy set to zero. This has the effect of creating a large differentiation in surface values between any neighbours to this zeroed voxel, which will bias the gradient vector away from it. A potential solution to this is instead of setting an intersected voxel's proxy to zero, set it to a value just below the current isosurface value. The downside to this is that if isosurface exploration is performed, this intersected voxel will then show up as part of a different surface.

By observing the third row of figure 5.4, we see the effects of moving the sphere too rapidly on the upper part of the hole: there are large portions of the surface remaining where the sphere passed through. Since the point shell contains points only on the sphere's surface, and not on its interior, fast movements of the shell will result in poor collision detection. The lower part of the hole illustrates what happens when the sphere is moved slowly through the surface.

The collection of graphs depicted in figures 5.5 - 5.8 represent 3 different sessions using the **half** dataset on cadomin. The 3 sessions are:

- **auto** The dataset was rotated at a constant speed, 3.5 degrees, per frame while the viewpoint was moved progressively closer to the model for 10 seconds, and then farther for another 10 seconds. The values for this session serve as a benchmark for comparison to the remaining sessions.

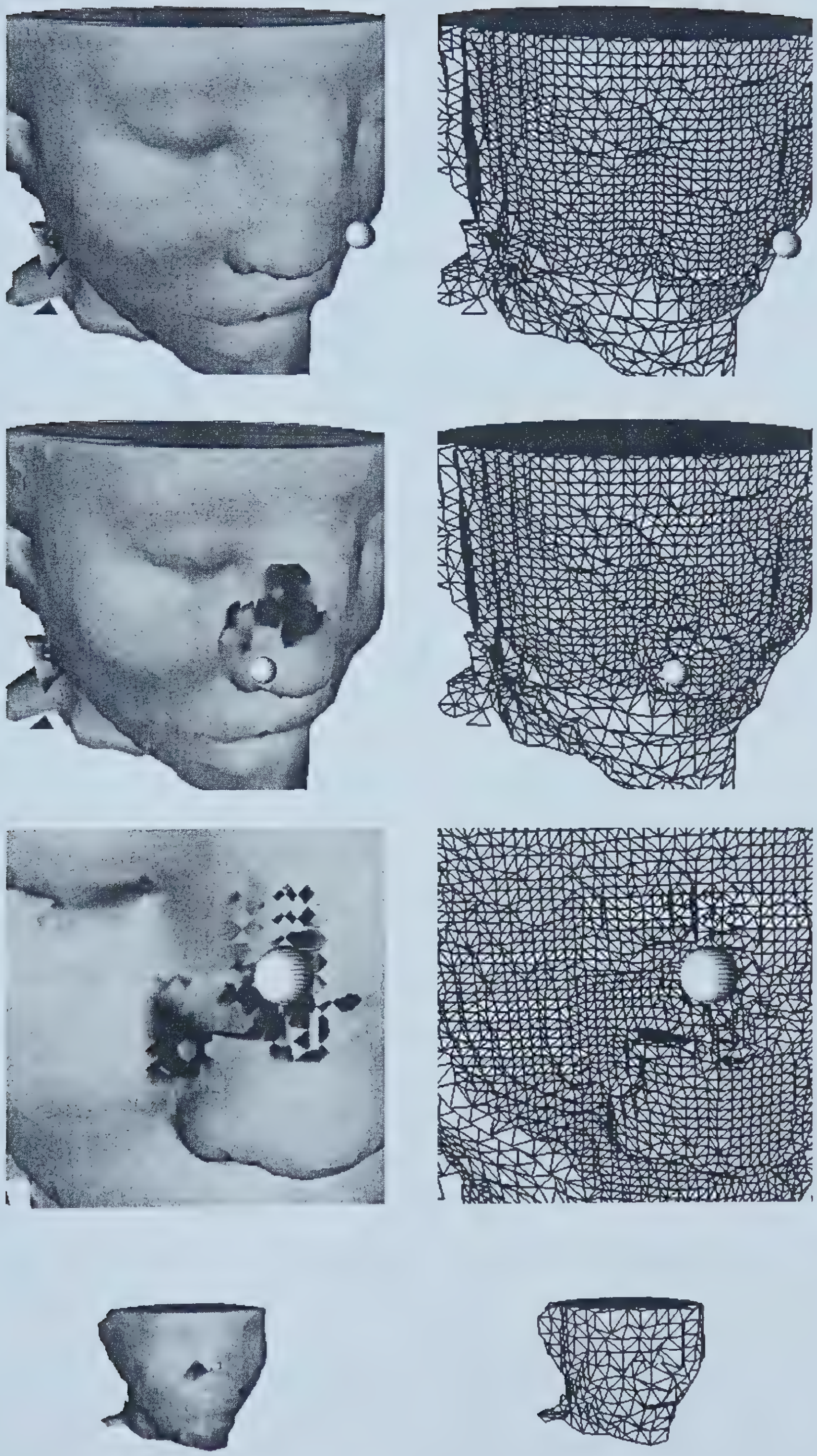


Figure 5.4: Sequence of collisions with the **half** dataset, from various viewpoints

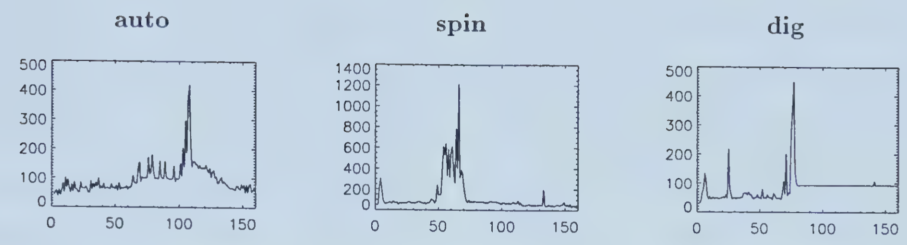


Figure 5.5: Frame times (ms) vs. frame number

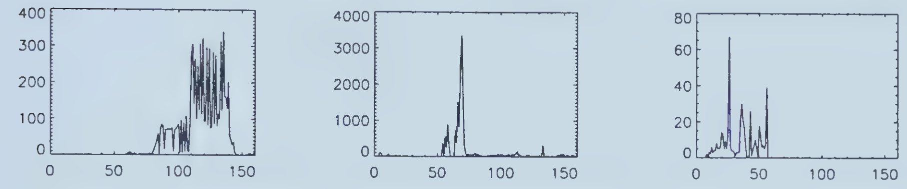


Figure 5.6: Fold counts vs. frame number

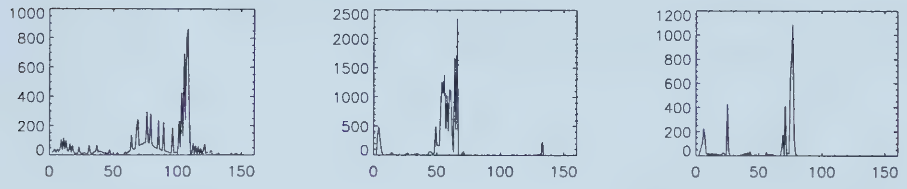


Figure 5.7: Unfold counts vs. frame number

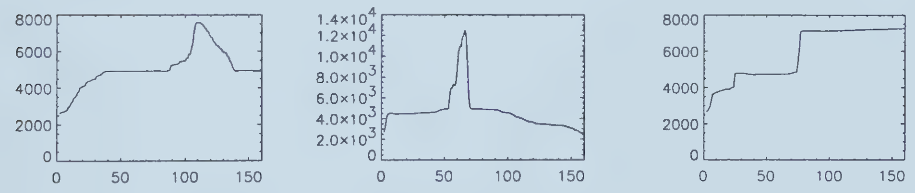


Figure 5.8: Triangle count vs. frame number

- **spin** The dataset was rotated arbitrarily in a fashion simulating the way a user would potentially view the **half** dataset. Again the viewpoint was moved progressively closer to the model for 10 seconds, and then farther for another 10 seconds.

- **dig** The viewpoint was moved progressively closer for 10 seconds, and then the sphere was used to scoop away at the surface.

An interesting relationship becomes apparent from these graphs: frame time correlates directly with the number of unfolds performed during that frame. This is most visible in the **auto** session: near frame 100 there is a large spike in folds, unfolds, and polygon count. Near frame 110 the unfolds drop off, but the folds and polygon count remain high. The frame time experiences a sharp decrease at frame 110, not quite returning to the level it was

at prior to frame 100. Following frame 110 the frame time decreases gradually, in a manner similar to the decrease experienced by the polygon count. This would indicate that polygon count also has a strong effect on frame time, though not as much as the number of unfolds.

The results from the **spin** session verify the relationship between the unfold count and frame time. Both the fold and unfold counts experience a small peak followed by a much larger peak, with the difference between the peaks being much greater for the fold count. Comparing frame time to this, we see that a much closer correlation is made with the unfold count peaks. Near the end of the session the frame time experience a very slight decrease which can be attributed to a corresponding decrease in the polygon count.

Analyzing the results from the **dig** session, we see a similar pattern emerge. A comparison of the unfold count versus the fold count is more difficult, as the disparity between the two is nearly a whole order of magnitude. The effect of an increased polygon count is visible however; the frame time is higher immediately after the large spike at frame 80, even though there are no folds or unfolds taking place. We also see a slight increase in the polygon count near the end of the session, a product of the cell repolygonization required after colliding with the surface.

Chapter 6

Future Work

6.1 Algorithmic Improvements

There are many algorithmic improvements and additions that we'd like to see added to HIME. These can be classified into improvements for speed, and improvements to visual accuracy.

6.1.1 Accuracy

In order to improve visual fidelity, SMC could be done away with and replaced with the classic MC algorithm. This would cause problems with vertex placement: referring all the way back to figure 3.6, MC interactively generates vertices along cell boundaries meaning that interpolation would need to occur along the dotted lines of the upper right figure. In effect, reinterpolation might be required with each proxy change. This would certainly cause a drop in frame rate for an unknown increase in the quality of the approximation.

A simple improvement could be made to the normal calculation. Currently, normals are generated for the voxel centres since it is assumed that this is where the vertices will lie. Once the vertices have been interpolated this is no longer the case. A similar interpolation could be done for the normals in order for them to accurately reflect the surface gradient at the given vertex's position. Another improvement to the normal calculation could be made for the surface regeneration after a collision. The number of surrounding neighbours used in the averaging scheme could be increased, thus improving the accuracy of the averaged normal.

Currently vertices are interpolated in order to lie closer to the surface. This could be replaced with a spring system to 'pull' the vertices closer to the actual surface resulting in a more continuous surface.

6.1.2 Speed

An additional step could be added to the redundant cell traversal scheme to avoid processing duplicate cells. If 2 adjacent voxels need to be updated throughout the course of a frame, their shared face will get processed twice: edge cells may be processed 4 times, corner cells up to 8 times. The current implementation prevents cells on this face from being passed through updateSurface() more than once – it will still subdivide the face twice in order to check if updateSurface() needs to be called. The time to update a large number of voxels would decrease if this second iteration was skipped altogether. A preliminary method we've come up with is a marking scheme by which unmodified voxels and voxels not needing updating are marked with 0, voxels in need of an update are marked with 1, and voxels that have been updated in the current frame are marked with 2. Updates would occur on cells for which all voxel proxies are marked with 1. This method is as yet untested. Another method to resolve this is to use a hash table to mark the cells visited during a frame; such a scheme is given in the appendix to [WG90].

A reduction in HIME's memory footprint would improve the performance on large datasets, and could be done without sacrificing performance in other areas. As discussed in chapter 5, representing the leaf nodes with a simplified structure would reduce the memory footprint by at least 30%. Other memory enhacements that could be made include improving the locality of reference to keep data in the processor cache as long as possible, and a memory map which would page out infrequently used data.

Another potential performance enhancement is the exploitation of parallelism. Distributing the work over multiple processors is an excellent strategy for speeding up any algorithm. HIME is an excellent candidate for parallelization, as it can be easily broken down into 2 basic tasks, Update and Render. Furthermore, the Update task could be broken down as the volume containing the surface is subdivided by the octree into non-overlapping subvolumes. As shown in chapter 5, the bottleneck in the system is the surface updating; a simple scheme would pass a subvolume to each available processor node. Care would need to be taken to prevent the Render process from rendering triangles in a subvolume that had not been completely traversed by the Update process. This would cause 'dropouts', or triangles that disappear for a frame before reappearing. Dropouts are the result of the Update process removing triangles from the render list, and the Render process sweeping through the volume before these triangles are replaced with their coarser or finer siblings.

6.2 Potential Applications

The system as presented is quite extensible, allowing for a large number of applications. One exciting use for HIME which has been mentioned throughout this document is the idea of

interactive isosurface exploration. Some preliminary work has been done on an implementation of HIME using queueing to perform surface updates. This enables incremental updates of the surface: if the current isovalue is modified, the entire tree is marked for updating and the update process gradually works its way down through the levels. More work is required before this becomes functional.

Currently vertices are statically offset from their original positions as part of the interpolation scheme. These offsets could be perturbed at run-time by an animation of some kind, for example a procedural animation, to give a certain effect. This was first demonstrated in [Mel98] where the surface was animated to appear organic, similar to a heartbeat.

For interactive sculpting, an extension could be made to the collision algorithm that filters the voxels rather than setting them to zero. An example would be a heat-gun application where the surface wouldn't disappear immediately, but gradually over time.

HIME could also be integrated with a haptics device and a collision-based system to study the physical accessibility of a model, as in [MPT99]. Instead of unfolding nodes that are closest to the viewpoint, those that are nearest the dynamic object controlled by the haptics device could be unfolded.

Chapter 7

Conclusion

HIME provides a framework for dynamically altering the surface of a polygonal model in real-time. The use of octrees in HIME allows hierarchical representation of large datasets, enabling the interactive display of a model that in its raw form would have been certainly been non-interactive during rendering.

7.1 Contributions

There are a number of contributions to current research by HIME. HIME presents a framework for dynamic interactive isosurface generation, into which any number of features can be inserted to provided different functionality. In the area of modifiable environments, HIME allows for a level of detail that's interactively variable; it also enables work with large datasets. In the area of isosurface generation, HIME presents a novel solution towards the elimination of cracks occurring in the generation of surfaces with variable levels of detail.

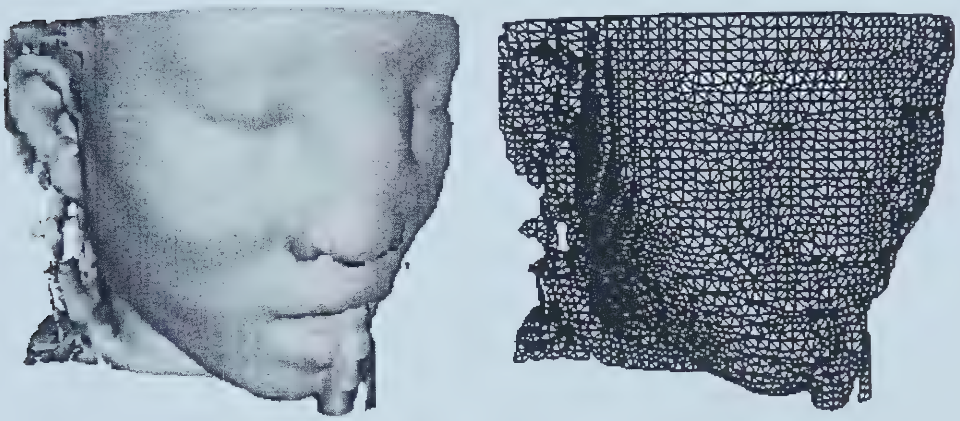
HIME presents an interactive viewpoint with sculpting in 3 dimensions, unlike [WK95] where sculpting is done from a static viewpoint on a 2-D image, with intersections mapping into 3-D using the z-buffer value as the depth. As well, the final result of the intersection is an image generated by a raycasting and compositing approach, whereas in HIME the final result is interactively viewable as a collection of polygons.

In comparison to [GH91], HIME allows fine-detail sculpting at any resolution with an interactive viewpoint no matter the current level, as opposed to allowing an interactive viewpoint only when using a coarse representation with coarse tools. The highest resolution rendered by [GH91] was 30^3, whereas HIME is able to interactively render a $256 * 256 * 93$ volume.

7.2 Review of Research Goals

Our goal of interactivity was acheived, with HIME easily rendering a $256*256*93$ volume at 15 FPS. Our goal of preserving visual quality of the model was also acheived, with the various

approximations generated by HIME remaining faithful to the original model. Figure 7.1 illustrates the quality of HIME's approximations. The top level of the figure illustrates the $128 * 128 * 93$ model rendered at a fixed resolution of $64 * 64 * 46$ with 31,000 polygons. The bottom portion of the figure illustrates the result calculated by HIME: the model was frozen and the viewpoint pulled back to demonstrate the changes in level along the surface. The viewpoint was zoomed in on the right eye and was brought close enough to the model for HIME to unfold down to the leaf level. The variable-level model is composed of 8,333 polygons. The fidelity of the area of interest, the right eye, as generated by HIME


Fixed-level, downsampled once from original dataset

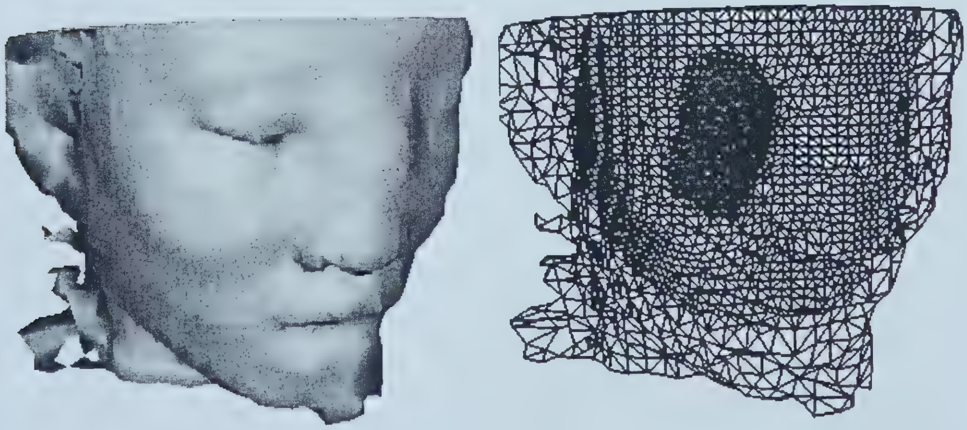

Variable-level calculated by HIME

Figure 7.1: Fidelity comparison

is actually better than that of the downsampled version even though HIME generates one third the amount of polygons. As such the goal of preserving visual fidelity was certainly acheived.

Unfortunately, our goal was to render a 512^3 volume interactively was not acheived. This was due to HIME's memory footprint being much larger than anticipated, an issue that can be resolved fairly easily with the knowledge that the structure containing octree leaf nodes can be trimmed down by roughly 30%. We feel that this is a conservative estimate, and with some careful analysis of the remaining structures a much larger reduction in memory usage could be had.

Appendix A

Screen-space Extent Calculation

A key element in HIME's surface update scheme is the ability to calculate the projected size of a voxel as it would appear on-screen. This size would then be used to determine if the voxel should be folded, unfolded, or left alone.

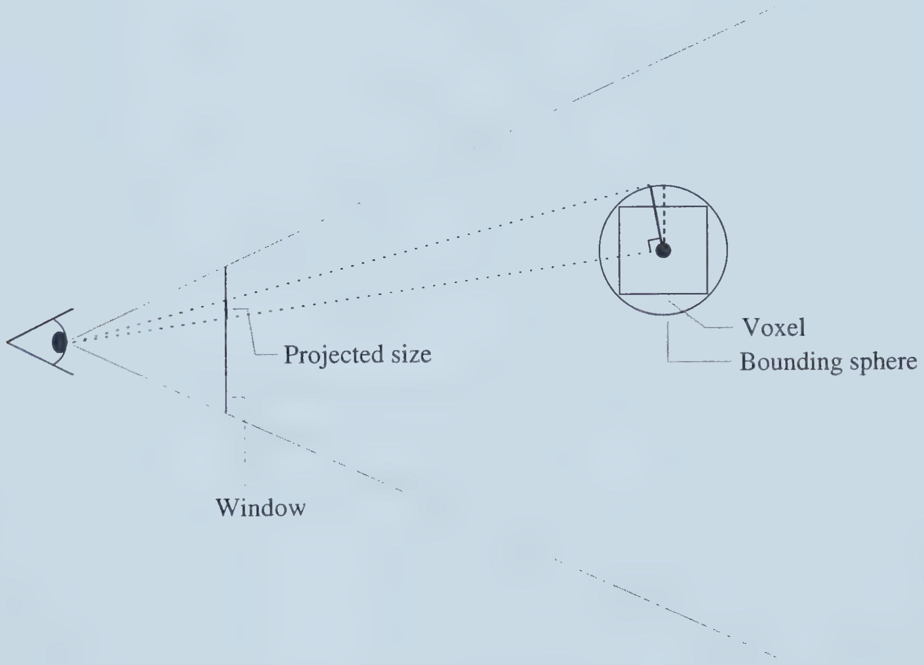


Figure A.1: Projected size of a voxel's approximated size

Figure A.1 illustrates the desired result of the extent calculation. All voxels in HIME have an associated bounding sphere which is used in place of the voxel itself.

There are 2 methods for calculating screenspace extent that were tested for HIME. The first was to use the graphics subsystem to determine the placement of the projected points, and then take the difference. The modelview matrix was multiplied by the projection matrix

to give a viewspace transformation matrix, M_v. The centre of the bounding sphere was multiplied by this matrix to get its position in the window. The second point corresponds to the point on the bounding sphere's surface that will maximize the projected screenspace extent, as shown in figure A.1. This point is equivalent to the intersection point of the vector orthogonal to the ray from the viewpoint to the sphere's centre. This point is geometrically difficult to compute, so to simplify the calculation we take the point at the intersection of a ray through the sphere centre parallel to the viewscreen, indicated by the thick dashed line in the figure. This simplification suffers from incorrectly calculating the screenspace extent when the sphere is near the boundary of a perspective projection view volume.

A second method is to use trigonometric principles to simplify the calculation. In addition, a given voxel knows it's level in the octree and from this we can obtain the radius of the bounding sphere: voxels at level l have a bounding sphere of radius 2^{L-l}. We assume that the viewpoint will always be at the origin, allowing a simple calculation to determine the distance from the viewpoint to the bounding sphere centre. If the viewpoint is not at the origin, the translation to move it there must be applied first. We then calculate the worldspace coordinates of the bounding sphere centre, which renders the distance calculation to the origin trivial. The screenspace extent is then taken to be the ratio of the sphere radius to the distance to the sphere centre. This method suffers from not being affected by the window size nor the field-of-view. This could be remedied by relating the calculated ratio to the ratio of the distance from the eye-screen distance of the viewer to the window size, and modulating the final result accordingly. As is, this method suffers from underestimating the value of the screen extent of the sphere when the sphere lies near the boundary of the viewing volume. This is not necessarily a bad thing, as it creates a form of a focus area about the centre of the screen: the voxels near the centre of the screen will thus be at a finer resolution than those at the fringes.

Appendix B

Glossary of Terms

Active cell list : list of cells used to polygonize the current frame. Created from the *active voxel list.*

Actual region : region defined by the BONO algorithm; the original volume

Active voxel list : list of voxels used to calculate the surface for the current frame. This list is traversed and the cells for each voxel in the list are polygonized, creating the surface.

Cell : cube formed by connecting the centre points of 8 adjacent voxels.

Conceptual region : region defined by the BONO algorithm for subdividing a volume; the volume occupying the smallest power of 2 in dimension along an edge that still encloses the original space

Detail metric : measure by which nodes in the octree are folded or unfolded each frame.

HDS : Hierarchical Dynamic Simplification. A system implemented by David Luebke by which polygonal information is stored in an octree, and nodes are folded or unfolded based on their projected size on the screen. Much of the work done in HIME was based on this system.

Isosurface : also referred to as *isovalue surface*; the surface created by displaying the values equivalent to the given isovalue.

Isovalue : in a volume dataset, any particular value contained in the dataset at which a surface may be generated.

MC : marching cubes [LC87].

SMC : simplified marching cubes [Mel98].

Subcell : cell completely enclosed by a coarse-level voxel

Voxel : a single unit of volume, created by dissecting a space into equal cubes. Each cube is represented by a value at its centre.

Voxel proxy : representative value used in the polygonization phase of HIME. This value is used to approximate a leaf-level voxel, regardless of the level of the voxel to which it belongs.

Bibliography

[BW00] D. Brodsky and B. A. Watson. Model simplification through refinement. In *Graphics Interface*, pages 221–228, 2000.

[CLMP95] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. *1995 Symposium on Interactive 3D Graphics*, pages 189–196, April 1995. ISBN 0-89791-736-7.

[Gar99] Michael Garland. Multiresolution modeling: Survey and future opportunities. In *Eurographics '99 Proceedings*, 1999.

[GH91] Tinsley Gaylean and John Hughes. Sculpting: An interactive volumetric modeling technique. In *SIGGRAPH 91 Conference Proceedings*. ACM SIGGRAPH, 1991.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Proceedings of SIGGRAPH 96*, pages 171–180, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[GWW86] I. Gargantini, T. R. S. Walsh, and O. L. Wu. Displaying a voxel-based object via linear octtrees. In *SPIE 626*, pages 460–466, 1986.

[Hop97] H. Hoppe. View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH'97*, pages 189–198, 1997.

[ILG96] Veysi Isler, Rynson Lau, and Mark Green. Real-time multi-resolution modeling for complex virtual environments. *Proceedings of VRST 96*, pages 11–19, July 1996.

[KBGT97] Mike Krus, Patrick Bourdot, Francoise Guisnel, and Guillaume Thibault. Levels of detail and polygonal simplification. *ACM Crossroads*, 3(4), 1997.

[KCVS98] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Nas-Peter Seidel. Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH 98 Conference Proceedings*, pages 105–114, 1998.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings*. ACM SIGGRAPH, 1997.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM TOG*, 9(3):245–261, July 1990.

[Lue97] David Luebke. A survey of polygonal simplification algorithms. Technical Report TR97-045, UNC, 1997.

[Mel98] Stan Melax. Using a binary voxel tesselation with spring edges to create a deformable rubbery 3d environment. http://www.cs.ualberta.ca/ melax/vox/vox.html, 1998.

[MPT99] William McNeely, Kevin Puterbaugh, and James Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *SIGGRAPH 99 Conference Proceedings*. ACM SIGGRAPH, 1999.

[MSS94] C. Montani, R Scateni, and R. Scopigno. Discretized marching cubes. In *Proceedings of Visualization*, pages 281–287, 1994.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990. ISBN 0-201-50255-0.

[SFYC96] R. Shekhar, E. Fayyad, R. Yagel, and J. Cornhill. Octree-based decimation of marching cubes surfaces. In *IEEE Visualization*, pages 335–342, 1996.

[SGI98] SGI. Recommended graphics tools for application developers, 1998. http://www.sgi.com/developers/nibs/1998/graphictools_nov98.html.

[WG90] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation extended abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 57–62, 1990.

[WK95] Sidney W. Wang and Arie E. Kaufman. Volume sculpting. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 151–156. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[WW92] Alan H. Watt and Mark Watt. *Advanced Rendering Techniques*. Addison Wesley, 1992. ISBN 0-201-54412-1.

[XESV97] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, June 1997.

[ZSS97] Denis Zorin, Peter Schröder, and Wim Sweldens. Interactive multiresolution mesh editing. *Computer Graphics*, 31(Annual Conference Series):259–268, 1997.